Ministère de l'Education Nationale

Université de Montpellier II
Facultés des Sciences

MASTER 2 INFORMATIQUE SPECIALITE A FINALITE PROFESSIONNELLE & RECHERCHE UNIFIEE

RAPPORT DE STAGE

effectué à

IBM Montpellier

du 6/04 au 30/09/2009
par

Renaud GUIOT-BOURG

Directeur de stage de l'entreprise

Jean-Michel RODRIGUEZ

Directeur de stage de l'université

Gérard SUBSOL

Analyse et développement d'outils permettant l'optimisation et la visualisation des données de pilotage d'un data center

REMERCIEMENTS

Je tiens à remercier, au travers de cette page, toutes les personnes qui ont permis que ma mission se déroule dans les meilleures conditions possibles en m'accompagnant durant toute la période du stage.

Au sein d'IBM:

Monsieur **Luc COLLEVILLE**, manager du service 'Dynamic Infrastructure Leader Center' qui m'a permis de m'intégrer au sein de l'équipe en me guidant dans les différents services.

Monsieur **Jean-Michel RODRIGUEZ**, mon tuteur officiel chez IBM, pour m'avoir permis de m'adapter rapidement au fonctionnement de l'entreprise en me présentant les aspects techniques du travail en équipe chez IBM.

Monsieur **Xavier VASQUES**, mon tuteur pour la partie HPC (High-Performance Computing), pour m'avoir guidé tout au long du stage en ce qui concerne la parallélisation et l'intégration du code sur les serveurs du HPC, et pour m'avoir aidé à prendre les décisions importantes quand elles s'imposaient. Merci aussi pour sa précieuse aide apportée tout au long du stage.

Monsieur **François BRIANT**, Distinguished Engineer, pour son soutient tout au long du stage, son suivi des avancées du stage et ses précieux conseils.

Monsieur **François-René ROUGEAUX**, mon tuteur pour la partie visualisation de l'application, pour m'avoir aidé dans toute la partie intégration Visualisation / HPC, notamment lorsqu'il a fallu faire communiquer une blade avec la salle immersive.

Au sein de l'équipe ICAR:

Monsieur **Gérard SUBSOL**, qui m'a guidé pour la compréhension du sujet de stage et pour son aide précieuse sur les descriptions des algorithmes de traitement d'images 3D et l'intégration des données du bâton percé.

Monsieur **William PUECH**, pour toutes ses réponses aux questions d'ordre technique sur les algorithmes de filtrage d'images.

Enfin, merci à toutes les personnes que j'ai pu rencontrer dans les bureaux de Montpellier, qui ont su rendre l'ambiance générale conviviale et agréable.

Sommaire

1. Introduction

Le but d'un stage est de découvrir la réalité de la vie dans l'entreprise au travers d'un sujet, ou mission, confiée à l'étudiant afin qu'il puisse mettre en pratique les différentes compétences et techniques qu'il a appris lors de sa scolarité. Il est souvent la première expérience d'un étudiant, qui va, tout au long de son stage, apprendre le fonctionnement d'une entreprise, les réalités économiques ainsi que l'indispensable besoin de compétitivité.

Dans le cadre d'une collaboration entre le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) et IBM, un sujet de stage autour de la visualisation d'images 3D de très grande taille a été monté, et j'ai donc hérité de cette mission. Le but était de pouvoir effectuer des traitements algorithmiques sur une image 3D puis d'utiliser un algorithme spécifique pour construire la surface d'une sous partie de l'image afin d'isoler la partie que l'on souhaite visualiser en détail. Tout cela devait être réalisé en se servant de serveurs multiprocesseurs d'IBM car sur un ordinateur standard le traitement est bien trop long a être effectué. Puis, l'affichage devait être fait dans la salle immersive d'IBM, une salle disposant d'un affichage haute résolution, grâce à quatre écrans chacun piloté par un serveur.

Dans ce contexte, j'ai donc implémenté des algorithmes de traitement d'image existants, puis les ai parallélisés afin de réduire le temps d'exécution et j'ai ensuite exporté le résultat vers la salle immersive pour visualiser le résultat final. En six mois, j'ai donc rempli la mission qui m'était confiée en explorant plusieurs domaines de l'informatique, qui sont retranscrits au travers de ce rapport.

1.1 Présentation des acteurs principaux du stage

1.1.1 IBM

IBM est une entreprise ondiale, avec une longue histoire. Nous essaierons ici de donner une vision suffisamment précise mais concise de la firme, en partant de son histoire et de sa position mondiale vers la description du site et du service où cette mission a lieu.

IBM dans le monde

La société IBM est créée en 1911 de la fusion par Charles R. Flint de l'International Time Recording Company, Computing Scale Company, et de la Tabulating Machine Company, elle est basée à New York et emploie à l'époque 1300 personnes (à l'époque elle se nomme Computing Tabulating Recording : CTR, jusqu'en 1924). La vocation originale d'IBM est la conception et la commercialisation de matériels informatiques et en particuliers d'ordinateurs centraux (*mainframes*). Cependant à plusieurs reprises la firme a choisi de se positionner sur de nouveaux

domaines :

- En 1981 IBM produit le premier PC, construit sur les microprocesseurs x86 d'Intel. Des ordinateurs personnels existent déjà depuis quelques années mais la principale particularité de l'IBM PC à l'époque est d'être conçu en utilisant des composants standards du marché (et donc non- IBM), à l'inverse de ses concurrents. IBM cependant abandonne en 1987 le standard PC aux « clones » (ordinateurs compatibles PC) en voulant créer sa propre architecture fermée. Aujourd'hui une immense majorité des ordinateurs personnels sont compatibles PC, et ce sont Windows et Intel qui déterminent chaque année les critères standard PC.

- En 1993 le choix est fait de développer des offres intégrées de produits et services. Le positionnement sur le secteur des services est un tournant majeur dans l'histoire d'IBM, et cela lui permet de tirer parti de ses forces, notamment en recherche et développement et de proposer au client une expertise et des solutions étendues : conseil matériel et logiciel, conseil métier et externalisation. Depuis 2002 et l'acquisition de la branche conseil de PricewaterhouseCoopers, IBM est devenu la première entité de conseil dans le monde entier.

- En 1997 la nouvelle stratégie annoncée « e-business » par IBM a des implications fortes au niveau métier, logiciel et matériel. Cette annonce représente pour beaucoup le premier signe sur les marchés boursiers de l'importance des conséquences du développement d'un réseau au niveau mondial.

Leader mondial dans le domaine des services et des technologies de l'information, IBM est aussi la première entreprise pour la vente de matériel et de services informatiques, pour la location et le financement d'équipements informatiques. La société est positionnée sur tous les secteurs de l'informatique, ce qui lui permet de proposer de nombreux produits de haute technologie. IBM Monde est divisée en quatre zones géographiques : Asia Pacific, EMEA (Europe, Middle East, Africa), Latin America, et North America. IBM possède la plus vaste entité de recherche mondiale dans le domaine des technologies de l'information et c'est la structure qui dépose le plus de brevets dans le monde, 2 fois plus que le total des brevets accordés à ses 4 plus proches concurrents : Oracle, BEA, Microsoft, Computer associates).

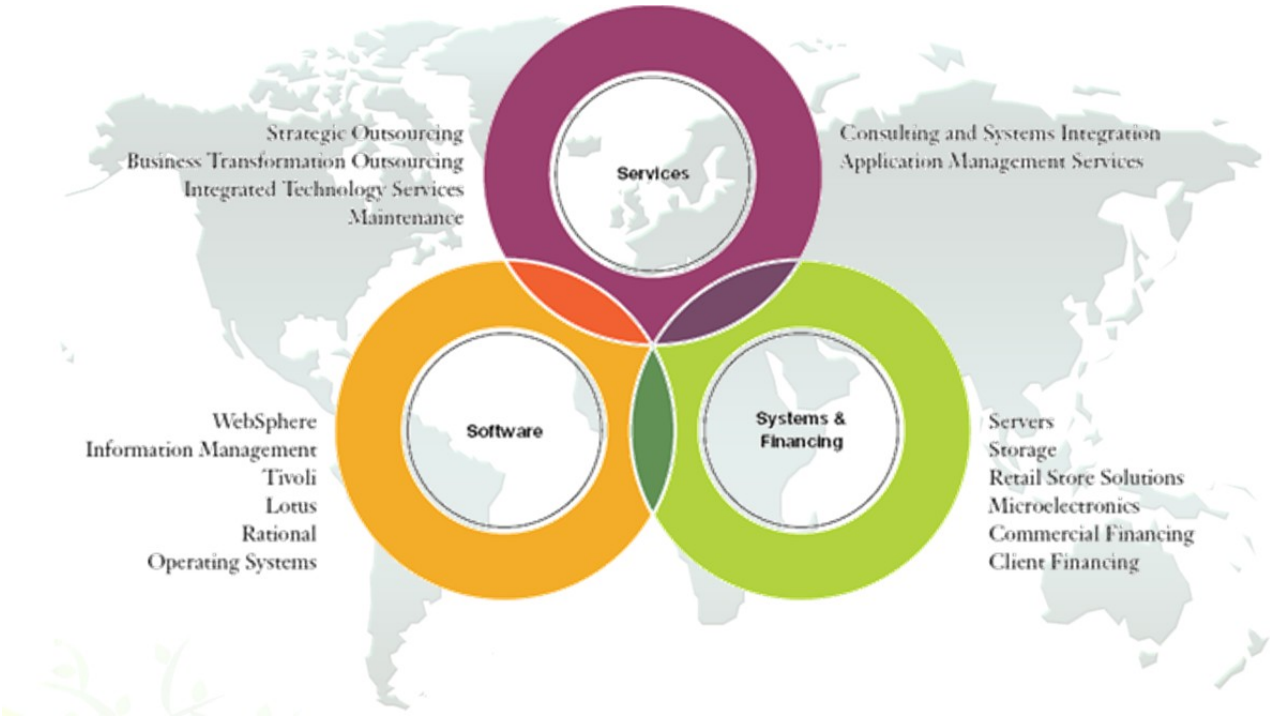


Figure 1. Pôles d'activité d'IBM

IBM France

IBM est implanté en France depuis 1914 et emploie aujourd'hui 11 000 collaborateurs. Daniel Chaffraix en est le Président-directeur général depuis mars 2007. Le siège d'IBM France est situé à Paris – La Défense. IBM France emploie 12500 collaborateurs dont 5200 personnes dans les activités de services. Ses activités commerciales et de services couvrent l'ensemble du territoire français avec six directions régionales, 12 agences commerciales et des sites de développement. Plusieurs sites en France sont positionnés au niveau mondial sur le plan de l'innovation:

- Pornichet et Toulouse en recherche
- Nice – La Gaude en innovation métier
- Montpellier en innovation technologique

IBM Montpellier

Comme IBM Monde, le site IBM Montpellier, créé en 1965, commence ses activités par une mission purement industrielle. Aujourd'hui le site IBM de Montpellier emploie environ 1000 personnes et même si la fabrication de serveurs d'entreprise haut de gamme reste une activité centrale, de nouvelles activités sont venues enrichir son activité de production:

- Tests de performance (*Benchmarks*)
- Valorisation des produits d'ancienne génération
- Infogérance
- Support technique pour les équipes de maintenance européennes.

Toutes ces activités sont déployées sur la zone Europe Moyen-Orient Afrique. Le « Product & Solution Support Center » est le centre du support EMEA (Europe Middle East Africa) pour l'ensemble des plates-formes IBM : *System z, System p, System i, System x et Total Storage*. Il représente la plus importante salle machine d'Europe. Il n'existe que trois centres de ce type dans le monde. Le PSSC fait partie de l'organisation *ATS* d'IBM (*Advanced Technical Support*), est équipé en permanence des dernières technologies et reçoit plus de 7000 visiteurs par an, en provenance de 60 pays différents. Le centre possède différents centres de compétences, son objectif est de fournir aux clients des présentations de produits ou d'architectures, des benchmarks, et des formations pour démontrer les avantages des systèmes et des infrastructures d'IBM. Au travers de ces différents centres de compétence, le PSSC :

- Offre à ses clients des briefings sur les dernières technologies IBM.

- Bâtit des solutions autours de l'infrastructure et des middlewares IBM.

- Met en oeuvre des Benchmarks et des preuves de concepts.

- Fournit du support technique aux équipes commerciales et technico-commerciales pour toute la région EMEA.

1.1.2 L'équipe ICAR

L'équipe ICAR et une équipe composée de jeunes chercheurs rattachés au Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM). Créée en 2005, elle a compté à ses débuts 4 permanents, puis jusqu'à 8 permanents pour se stabiliser de nos jours à 5 permanents: W. PUECH, G.SUBSOL, M.CHAUMONT, O.STRAUSS et F.COMBY. Ainsi, depuis fin 2008, l'équipe s'est stabilisée autour de ces 5 membres dont les domaines de compétences s'articulent autour de trois axes principaux associant image et interaction pour la manipulation des données virtuelles telles que les images, les vidéos et les objets 3D :

- Codage et protection
- Analyse et traitement
- Modélisation et visualisation

Comme tout laboratoire de recherche, elle s'appuie aussi sur des doctorants, environ une douzaine qui participent activement à la recherche et donc au développement du laboratoire. Ainsi, depuis 2005, pas moins d'une centaine de publications ont été crées, dont plus de la moitié ont été présentées lors de congrès internationaux. L'équipe ICAR s'est impliquée au niveau régional dans le projet régional INTS (Intelligence Numérique et Technologie Sensible) et apporte une aide et un soutien à de nombreuses entreprises innovantes régionales pour des applications directes des

recherches effectuées dans l'équipe. Les collaborations avec des entreprises se sont concrétisées entre 2005 et 2008 par 9 contrats dont 3 thèses CIFRE.

Après cette brève introduction, nous allons maintenant détailler un peu plus les trois domaines principaux de compétences de l'équipe.

Analyse et traitement (AT): l'axe AT s'intéresse à de nouvelles techniques de traitement bas-niveau de l'information permettant de représenter, dans un même cadre théorique, l'imprécis, l'incertain et l'incomplet, qui sont les types d'erreur en traitement des données.
Les recherches de l'axe AT ont principalement porté, entre 2005 et 2008, sur le développement de nouvelles approches en traitement du signal et des images en s'appuyant sur une extension de l'approche linéaire classique. Cette nouvelle approche permet de prendre en compte un défaut de connaissance du système utilisé, de mieux modéliser la représentation discrète d'un problème continu, de considérer les données numériques comme imprécises en valeur (quantification) et en position (échantillonnage) en plus de leur aspect aléatoire et enfin de quantifier l'erreur d'estimation induite par la propagation de l'erreur de mesure au travers des calculs. Ces recherches, outre leur aspect théorique, ont porté sur l'utilisation de cette nouvelle approche dans les domaines suivants: vision omnidirectionnelle, tomographie d'émission, estimation statistique, filtrage, transformation et analyse d'images, analyse du mouvement projeté et fusion d'informations géoréférencées.

Codage et protection (CP): l'axe CP s'intéresse à la transmission et l'archivage sécurisés de données visuelles. Cette protection peut être assurée soit par tatouage soit par cryptage et doit être robuste à la compression.
Le transfert, la visualisation et l'archivage de données visuelles sont des services numériques qui connaissent une forte croissance depuis 10 ans. Le développement de ce type de services soulève un nombre conséquent de problèmes non résolus à ce jour. Un premier problème concerne le temps de transfert ou d'accès. En effet, afin de réduire ces temps de transfert, les données visuelles doivent être comprimées. En fonction des applications la compression pourra être plus ou moins importante, réversible ou non. Un deuxième problème concerne les aspects sécurité, englobant les problèmes de confidentialité, d'intégrité des données, de traçabilité mais aussi de correction d'erreurs et de robustesse aux attaques bienveillantes ou non. Afin de résoudre le problème de sécurité, il faut souvent qu'une partie des données soit rendue complètement ou partiellement illisibles, non déchiffrable ou accessible uniquement à des personnes autorisées. Dans le contexte de cette problématique, l'axe CP a travaillé dans la sécurisation des données visuelles par insertion de données cachées et par cryptage robustes à la compression. Entre 2005 et 2008, l'axe CP de l'équipe ICAR a développé de nouveaux algorithmes combinant insertion de données cachées, cryptage et compression des données visuelles.

Modélisation et visualisation (MV): l'axe MV a pour objectif de traiter des données visuelles 3D. Il s'agit alors de modéliser ces données, consistant à en extraire une représentation synthétique afin de supprimer des défauts tel que le bruit d'acquisition, à déterminer la frontière de certaines régions et à extraire des amers pour caractériser certains paramètres comme la forme, et pour comparer différents échantillons entre eux en les recalant. La modélisation pourra aussi permettre de simplifier les données originales (en remplaçant par exemple un ensemble de triangles coplanaires par une équation linéaire) ou de visualiser l'objet en multirésolution (qui variera en fonction de la distance à l'objet par exemple). Les recherches au sein de l'axe MV ont été menées suivant les thèmes suivants: extraction de caractéristiques géométriques 3D, segmentation hiérarchique de nuages de points 3D non structurés et morphométrie 3D (l'idée est d'utiliser soit les caractéristiques géométriques 3D, soit la segmentation en régions pour essayer de caractériser la

forme d’un maillage).

1.2 Objectifs détaillés du stage

La visualisation et le traitement d'objets 3D est un problème classique dans l'informatique moderne. Ainsi, depuis plusieurs années, le traitement algorithmique d'images 2D ou 3D ne cesse d'évoluer: reconnaissance de formes, filtrage, tatouage et beaucoup d'autres techniques apparaissent au fur et à mesure que les recherches avancent. Dans certains cas, un ordinateur standard peut être suffisant pour exécuter de telles méthodes, notamment lorsqu'il s'agit de données 2D (photos ou images) car le volume de données à traiter est assez petit pour ne pas demander une grosse puissance de calcul. Cependant, et principalement lorsque les données à traiter sont en trois dimensions, le volume de données peut très vite augmenter et le temps de calcul ne sera plus du tout satisfaisant dans le cas où l'on utiliserait un ordinateur standard.
Ainsi, ce stage a pour objectif de se servir de serveurs de calculs multiprocesseurs dont dispose IBM afin de réduire au maximum le temps d'exécution de divers traitements algorithmiques effectués sur de grosses quantités de données pour se rapprocher au maximum du temps réel lors de l'exécution finale. Il m'a donc été permis tout au long de ce stage d'étudier les différents dispositifs d'IBM, afin de me servir de cette architecture pour améliorer le temps de calcul de l'application que j'avais à développer. Comme nous le verrons, le temps de calcul a pu être réduit en grande partie grâce à la parallélisation, dont les résultats sont tous présentés dans les sections **4.1** Résultats obtenus et **5.1.5** Benchmark.

Les données 3D et les utilisations de ce type d'application peuvent être diverses et variées, allant de l'imagerie médicale à la valorisation du patrimoine, en passant par la vidéoconférence. Par exemple, un médecin effectuant une IRM sur un patient pourrait utiliser cette application afin de récupérer le résultat de l'IRM, puis isoler un organe malade et ainsi pouvoir l'observer. On pourrait aussi imaginer, à partir de relevés de températures, obtenir une représentation d'une zone de températures, par exemple celles inférieures à 20°C, paramètre que l'on pourrait modifier afin d'afficher les déperditions de chaleurs. On obtiendrait ainsi une cartographie des températures modifiable en temps réel. Pour ce stage, il a été décidé de travailler sur un type de donnée particulier: comment traiter et visualiser des données 3D obtenues par microtomographie. Comme dit précédemment, actuellement il n'est pas possible d'effectuer le traitement sur l'image entière (dont les dimensions sont de 2780 x 836 x 908) à partir d'ordinateurs personnels standards. Sur des ordinateurs standards, le travail est donc effectué sur des données sous-échantillonnées ou sur une partie des données. Là est tout l'intérêt d'utiliser la grosse puissance de calcul des serveurs d'IBM, afin de pouvoir travailler sur l'intégralité des données.

Afin de mieux expliciter les objectifs de ce stage, nous allons maintenant voir une rapide présentation des trois étapes principales du stage:

- *Implémentation d'algorithmes de traitement d'image*

Il existe une quantité d'algorithmes qui effectuent divers traitements sur une image. J'ai donc choisi d'implémenter dans un premier temps deux algorithmes de traitement d'images qui permettent d'améliorer le rendu visuel d'une image: le filtre médian et le filtre gaussien. Tous deux sont des

algorithmes qui permettent de restaurer la qualité d'une image qui serait dégradée, à cause par exemple d'une mauvaise qualité des capteurs d'acquisitions de l'image 3D ou des dégradations ayant pour effet une perte de données.
Les données issues du bâton percé (qu'il fallait intégrer au code de l'application, ceci n'étant pas une sous-étape triviale) devaient être visualisées correctement. Pour cela, à partir du tableau de données fournit par les fichiers de type ANALYZE contenant les intensités en chaque voxel de l'image, il fallait reconstruire l'image 3D fin d'en visualiser le bâton percé. Dans cette optique, l'algorithme des Marching Cubes a dû être implémenté, puis le résultat de cet algorithme (des triangles mis bouts à bouts, voir section **5.1.2** Extraction d'iso-surface) devait être affiché en 3D.

- *Parallélisation*

La parallélisation est un outil très efficace pour peu que l'on dispose d'une architecture machine qui nous permette de s'en servir. La parallélisation a alors pour but d'exploiter au maximum les ressources qu'offrent le ou les serveurs afin de réduire au maximum le temps de calcul. Techniquement, cela se traduit par des directives qui sont insérées dans le code écrit lors de l'étape précédente afin de le rendre exécutable par plusieurs tâches, voir plusieurs serveurs. Toutes les tâches et serveurs communiquent alors entre eux pour se répartir le travail, ce qui permet d'optimiser les ressources disponibles.

- *Affichage (couplage calculs / visualisation 3D)*

La dernière partie du stage était sans doute la plus importante. En effet, il est intéressant d'effectuer des calculs sur l'image, mais cela ne sert à rien dans une application de ce type si on ne possède pas d'outils pour visualiser le résultat.
Lors de ce stage, il été donc fixé comme objectif l'affichage du bâton percé dans la salle immersive que possède IBM (4 écrans pilotés chacun par un serveur), après avoir effectué la partie calcul sur les serveurs multiprocesseurs. Il été donc nécessaire de faire communiquer les deux parties (serveurs de calculs et salle immersive) afin qu'elles travaillent en coopération pour un résultat final qui tend toujours vers un même but: l'affichage et le traitement des données en temps réel.

1.3 Planning prévisionnel

Ce stage se déroulant d'avril à septembre, il était indispensable de répartir le temps disponible entre les trois étapes principales afin de pouvoir me fixer des objectifs à atteindre et ne pas perdre de temps. En effet, six mois peuvent paraître un délai suffisamment long pour traiter une problématique donnée mais cela passe très vite quand on a un projet à mener à terme. Ainsi, je me suis fixé des objectifs pendant toute la période de ce stage, et ceux-ci sont retranscrits dans le diagramme de Gantt ci dessous:

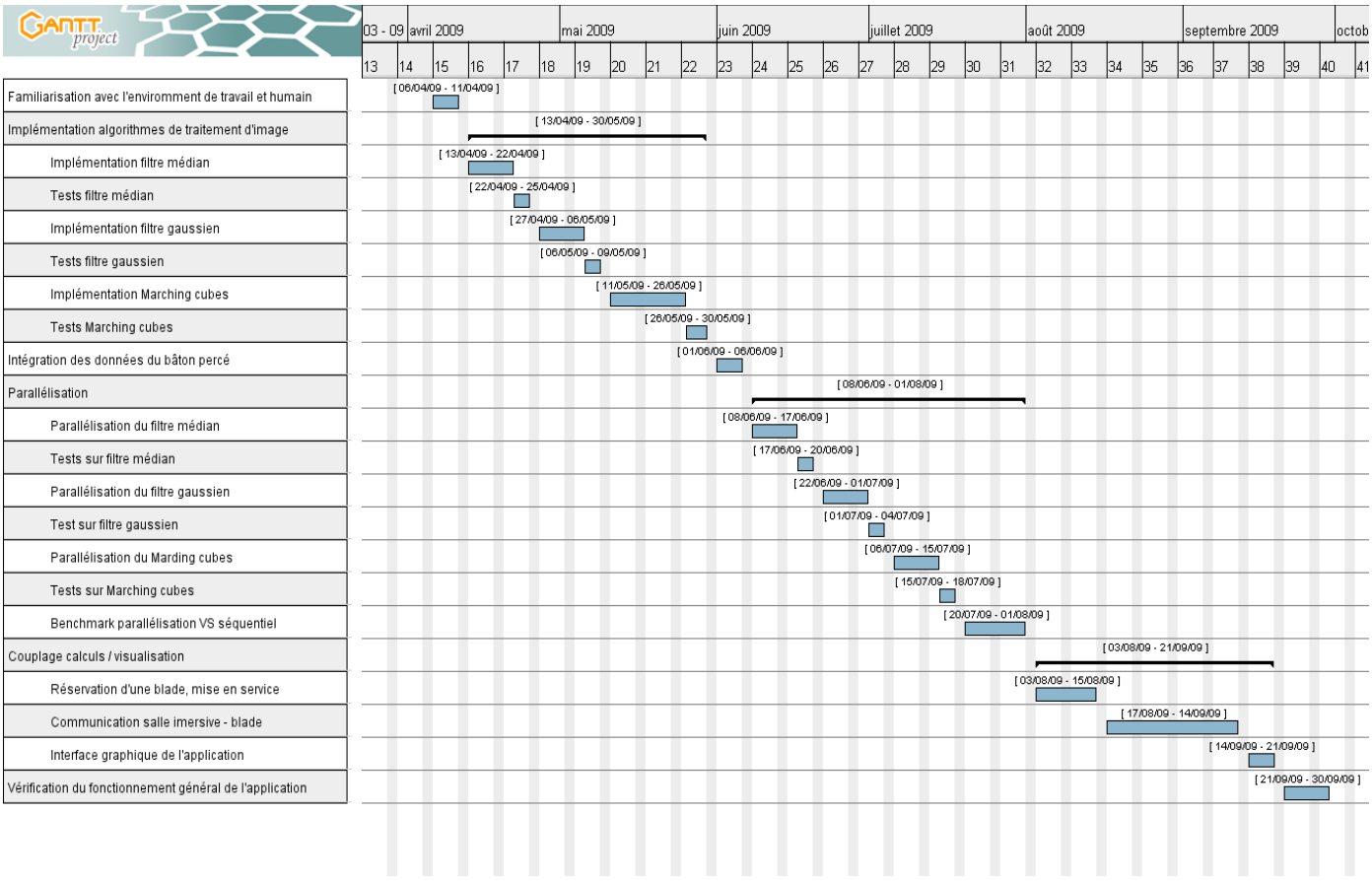


Figure 2. Planning prévisionnel des tâches à accomplir

2. Le problème – Méthodologie – Outils

2.1 Problématique

La problématique sur laquelle se fonde ce stage est assez simple: comment traiter et visualiser des objets en trois dimensions de très grande taille. C'est une question qui revient souvent lorsqu'on manipule des données 3D, car les images traitées sont issues de capteurs très précis qui fournissent donc une quantité non négligeable de points dans l'espace afin d'en représenter la forme voulue. Il est donc très intéressant de travailler sur ce type de problème car il peut être très formateur:

- Sur le point de l'algorithmie avec des algorithmes à implémenter qui sont beaucoup utilisés autant pour les images 3D que les images 2D classiques.

- Sur la parallélisation qui est un domaine qui va être de plus en plus utilisé dans le monde de l'entreprise avec la multiplication des ordinateurs multiprocesseurs à mémoires partagées.

- D'un point de vue plus personnel, sur la 3D car c'est un domaine très attrayant.

2.2 Méthodologie adoptée et outils utilisés

Méthodologie

Afin d'être plus efficace dans mon travail, j'ai décidé d'adopter une méthodologie « étape par étape » , c'est à dire que je me concentrais, parmi les trois phases du projet décrites ci-dessus, sur une seule à la fois.

J'ai donc commencé par implémenter les algorithmes de traitement d'image. Et j'ai dissocié les phases « implémentation du filtre médian » et « implémentation du filtre gaussien », dans le sens où je ne suis passé à la deuxième citée (filtre gaussien) qu'après avoir été sûr que la première (filtre médian) fonctionnait correctement. J'ai choisi d'implémenter d'abord le filtre médian car il était plus simple à comprendre et plus facile à implémenter que le filtre gaussien, ainsi cela me permettait de me réhabituer au langage C. De même, j'ai ensuite implémenté le Marching cubes car c'était le plus complexe des trois algorithmes, et j'ai ensuite testé son fonctionnement pour être certain qu'avec l'affichage grâce à OpenGL j'obtenais bien le résultat désiré.

Afin d'effectuer tous les tests nécessaires pour vérifier le bon fonctionnement de ces algorithmes, j'ai pris comme image à traiter une image virtuelle. En effet, je remplissais mon tableau

correspondant aux intensités en chaque voxel grâce à la fonction de la forme $f(x,y,z) = \frac{x^2}{a} + \frac{y^2}{b} + \frac{z^2}{c}$ où *a, b* et *c* sont des constantes. Ainsi, je pouvais donner à mon image les dimensions que je voulais, et cela me permettait de lancer mes algorithmes sur une image assez petite pour que les algorithmes s'exécutent en temps raisonnable.

Puis, je suis rentré dans la phase « parallélisation », avec d'abord un profiling du code puis insertion de directives OpenMP. J'ai commencé par le filtre médian, car il était assez simple et me permettait de bien comprendre le fonctionnement d'OpenMP. Je suis ensuite passé au filtre gaussien, un peu plus complexe selon moi que le filtre médian, pour finir par le Marching cubes qui était le moins simple de tous nécessitant l'ajout de verrous pour protéger certaines variables, comme nous le verrons section **3.3**. Cela m'a permis de découvrir peu à peu la parallélisation via OpenMP, et de progresser étape par étape dans un domaine que je ne connaissais pas avant mon stage.

Ce n'est qu'ensuite que je me suis attelé à la communication salle immersive / serveurs de calculs, car j'avais besoin à la fois d'un résultat à afficher en 3D mais aussi d'installer les bonnes librairies et compilateurs sur mon serveur.

Outils adoptés

La première interrogation concernait le langage dans lequel l'application serait développée. Après un petit temps de réflexion, il est apparu évident que je développerait en C, car ce langage est très proche du langage machine et permet notamment la manipulation de pointeurs et la gestion des ressources utilisées (en particulier lorsqu'il s'agit de réserver une taille mémoire en fonction du type de données que l'on a, très utile pour un projet dont les ressources demandées sont importantes). De plus, il permet aussi d'inclure des libraires telles que OpenGL et OpenMP, deux des librairies les plus utilisées dans le cadre de ce stage.

Comme je devais implémenter des algorithmes de traitement d'images déjà existants, je n'ai donc inclus aucune librairie de traitement d'image, qui peuvent cependant être téléchargées sur internet et dont certaines sont gratuites. Cependant, je me suis grandement inspiré de sources, cours et autres descriptions obtenues sur internet afin de comprendre l'utilité et le fonctionnement des algorithmes [1] [2] [3] [4].

De plus, il me fallait mettre en évidence le gain de temps obtenu grâce à la parallélisation, afin de pouvoir comparer les temps d'exécution des algorithmes implémentés. Dans ce sens, j'ai donc été amené à réaliser un Benchmark de l'application pour pouvoir expliciter la réduction du temps de calcul d'une part, mais aussi le temps de calcul des algorithmes en fonction de la taille des données et du serveur de calcul utilisé. L'explication des résultats obtenus pour ce Benchmark est disponible section **4.1** Résultats obtenus, et le Benchmark lui-même peut être visualisé dans la partie **5.1.5** Benchmark.

Enfin, comme dit précédemment, j'avais besoin pour mener à bien la mission qui m'était confiée d'un ensemble de librairies qui sont énumérées ci dessous:

- OpenGL (Affichage 3D)
  - glut (OpenGL Utility Toolkit)

- OpenMP (calcul parallèle sur architecture à mémoire partagée)

- GTK (pour l'interface graphique)

3. Synthèse de la solution apportée

3.1 Les données

En premier lieu, pour pouvoir se servir de ces algorithmes, il fallait que l'application possède un jeu de données sur lesquelles effectuer le traitement, c'est à dire une image en trois dimensions stockée en machine comme un tableau à trois dimensions. Pour cela, j'ai donc été amené dans un premier temps à développer un fichier C nommé Donnees.c qui permet de remplir ce tableau, lequel est ensuite accessible par tous ceux qui ont besoin des intensités en chaque voxel de l'image.

Afin que l'image à traiter ne soit stockée qu'une seule fois en mémoire, et dans le but de ne pas surcharger celle-ci en dupliquant l'image, j'ai donc déclaré le tableau à trois dimensions représentant l'image à traiter comme un pointeur. De fait, ce dernier est partagé par tous les fichiers C, ce qui permet à chaque fonction d'accéder à l'image grâce à l'adresse de ce pointeur, sans avoir un quelconque traitement à faire. Le fichier Donnees.c contient donc les fonctions qui se chargent de remplir ce tableau, et de déclarer les ressources mémoires qui nous sont nécessaires. L'application possède ainsi toutes les informations nécessaires au traitement de l'image:

- Tailles de l'image en X, Y et Z.
- Pointeur vers un tableau contenant les intensités en chaque voxel.

Au début, j'ai donc rempli ce tableau « à la main » grâce à une image virtuelle, appelée ainsi car remplie en se basant sur la fonction mathématique suivante:

$$f(x,y,z) = \frac{x^2}{a} + \frac{y^2}{b} + \frac{z^2}{c}$$ où *a, b* et *c* sont des constantes.

L'avantage de cette image virtuelle réside dans le fait que je pouvais changer les dimensions sans problème, j'ai ainsi pu partir d'une image de petite taille afin de tester le bon fonctionnement des algorithmes implémentés, pour évoluer vers une image de plus grande taille afin de réaliser le Benchmark.

Par la suite, j'ai crée dans le fichier Donnees.c une méthode permettant de remplir ce tableau à trois dimensions à partir du fichier ANALYZE contenant les intensités en chaque voxel issues de la microtomographie du bâton percé, ce qui permettait de travailler sur des données concrètes. Pour cela, j'ai pu compter sur un outil très appréciable, MicroView [5], qui permet de visualiser ce type de format de données 'en statique', c'est à dire qu'aucun traitement ne peut être appliqué sur l'image grâce à lui. Il m'a surtout permis de regarder l'intensité en chaque voxel de l'image, afin de comparer avec les valeurs que j'obtenais lors de la lecture du fichier, et donc vérifier que les valeurs obtenues étaient correctes.

Ainsi, pour obtenir des valeurs cohérentes à partir du fichier de type ANALYZE, il fallait procéder de la manière suivante:

- Déclarer un pointeur sur une chaîne de caractères. Sa taille devait être exactement du nombre d'intensités contenues dans l'image (soit la dimension de l'image en X multipliée par celles en Y et en Z) multipliée par deux car les valeurs sont codées sur deux octets.

- Déclarer un pointeur sur un fichier

- Lire le fichier ANALYZE grâce à la fonction fread qui prend comme paramètre le pointeur sur une chaîne de caractère déclaré auparavant. Celui-ci contiendra alors, après exécution de la fonction, l'ensemble des valeurs composants l'image. Cependant, elles sont codées sur deux octets, il faut donc passer par une dernière étape.

- Lire les éléments contenus dans le pointeur sur la chaîne de caractères deux par deux en les « castants » afin de les déclarer comme « signed short » (car les valeurs obtenues ne dépassent pas ± 10000). Pour récupérer la bonne valeur, il faut alors multiplier la deuxième valeur obtenue par 256 et additionner le tout à la première valeur obtenue.

- La valeur obtenue correspond alors à l'intensité du point (0,0,0); puis les valeurs suivantes sont mises dans le tableau en remplissant celui-ci selon l'axe X, puis Y et enfin Z.

Les données étant maintenant disponibles, j'ai donc pu lancer mes premiers travaux concernant l'algorithmie, et plus particulièrement le filtrage d'image avec le filtre médian et le filtre gaussien. Après un délais de compréhension de ces algorithmes, je me suis donc attelé à leurs implémentation.

3.2 Algorithmes de traitement d'images

3.2.1 Filtrage d'images

Une fois les données disponibles, indispensables au fonctionnement des algorithmes, je suis donc passé à l'implémentation des filtres médians et gaussiens. Ainsi, j'ai donc créer les deux fichiers C Median.c et Gaussien.c correspondants respectivement au filtre médian et au filtre gaussien. Ces deux implémentation sont semblables dans le sens où elles se servent des intensités en chaque voxel de l'image à traiter, puis effectuent des traitements en fonction de ces valeurs et renvoient un pointeur sur l'image ainsi modifiée. Ces deux fichiers ne diffèrent donc que dans la manipulation des valeurs récupérées, car elles utilisent exactement les mêmes paramètres et renvoient toutes les deux une image modifiée.

Le tout premier algorithme implémenté fût donc le filtre médian, pour lequel j'ai décidé de construire une structure, de la même manière que pour le remplissage des données, avec l'image et sa taille. Ainsi, lors de la récupération de l'image, il m'a suffit déclarer mon image et d'aller la

chercher dans le fichier Donnees.c qui contenait l'image initiale. Ensuite, il m'a fallu déclarer une autre image de même taille (donc un pointeur correspondant au tableau à trois dimensions des intensités en chaque voxel), de le remplir avec les bonnes valeurs suite aux calculs effectués sur l'image d'entrée, et de retourner ce pointeur qui correspond donc à l'image après traitement. Bien sur, dans un souci d'optimisation, la taille mémoire réservée pour l'image originale est ensuite libérée car il ne sert plus à rien de stocker cette dernière. La seule image utile est l'image après traitement.
L'algorithme du filtre médian utilise une structure de tri afin de pouvoir extraire la médiane (cf section **5.1.1.2** Filtre médian). Ainsi, le choix d'un algorithme de tri est une étape qu'il ne faut pas négliger car selon celui que l'on utilise la complexité du filtre médian varie. Ainsi, dans un premier temps, j'ai implémenté l'algorithme de tri par bulle puis suis passé à l'implémentation d'un tri nommé « quicksort » car c'est un algorithme de tri particulièrement efficace. La complexité du filtre médian s'en trouvait donc réduite, chose indispensable car, comme nous le verrons section **5.1.5** Benchmark, c'est celui qui est le plus lent à l'exécution.

En ce qui concerne l'implémentation du filtre gaussien, j'ai donc pu me baser sur ce que j'avais déjà écris pour le filtre médian, car les deux algorithmes sont assez ressemblants, comme décris plus haut. La principale difficulté fût donc de comprendre son fonctionnement, car pour l'ossature du code cela ressemblait très fortement au filtre médian hormis les traitements effectués à partir de l'intensité des voxels.

Une fois ces étapes franchies, je disposais donc de deux fichiers avec une seule méthode utile, celle qui permet de lancer le traitement sur l'image. Ainsi, dans mon fichier principal, il me suffisait de faire appel à l'une de ces deux méthodes (ou les deux à la fois) présentent dans ces fichiers afin de traiter mon image et de renvoyer proprement une image filtrée. Après chaque implémentation, j'ai donc utilisé l'image virtuelle avec une petite taille afin de tester le bon fonctionnement des filtres. Le temps de calcul étant alors relativement faible, cela me permettait de lancer plusieurs tests afin d'être véritablement certain que les calculs effectués étaient corrects.

L'implémentation de ces algorithmes fût une étape importante, car elle m'a permis de me refamiliariser avec le langage C afin d'être dans de bonnes dispositions pour l'implémentation du Marching cubes, car c'est de loin le plus complexe des trois algorithmes que j'ai du implémenter lors de ce stage.

En parallèle, j'ai écrit un Makefile, fichier qui permet de compiler les fichiers contenant le code de l'application avec les bonnes options de compilation et les bons chemins d'accès aux librairies utilisées. Pour cela, je me suis servie d'un très bon tutoriel disponible sur internet [6]. Un Makefile permet donc de lancer la compilation sur n'importe quelle plateforme en modifiant simplement le répertoire qui contient les librairies, et en lançant le programme make à partir du dossier contenant le code. Ceci apporte un avantage de portabilité évident. Un Makefile peut aussi servir, par exemple, pour l'archivage de document ou la mise à jour de sites.
Ainsi, lorsque j'ai exporté mon code vers les serveurs multiprocesseurs du HPC, il me suffisait pour compiler de me servir du programme make plutôt que d'avoir à écrire une commande pour chaque fichier. Après inclusion de toutes les librairies, voici à quoi ressemble mon Makefile:

```
COMPIL = gcc
EXEC = Marching
CFLAGS = -Wall

OBJ = obj/
SRC = src/
INC = inc/

XLIBS = -L /usr/lib64
LIBS = $(XLIBS) -lglut -lm
GPROF = -g -pg
PARA = -fopenmp #-openmp

all: $(EXEC)

Marching: Pile.o Affichage.o Donnees.o Gaussien.o Median.o Marching.o Affichage3D.o main.o
	$(COMPIL) -o bin/$(EXEC) $(OBJ)Pile.o $(OBJ)Affichage.o $(OBJ)Donnees.o $(OBJ)Gaussien.o $(OBJ)Median.o $(OBJ)Marching.o $(OBJ)Affichage3D.o $(OBJ)main.o $(LDFLAGS) $(LIBS) $(PARA)

Affichage3D.o: $(SRC)Affichage3D.c
	$(COMPIL) -o $(OBJ)Affichage3D.o -c $(SRC)Affichage3D.c $(CFLAGS)

Pile.o: $(SRC)Pile.c
	$(COMPIL) -o $(OBJ)Pile.o -c $(SRC)Pile.c $(CFLAGS)

Affichage.o: $(SRC)Affichage.c
	$(COMPIL) -o $(OBJ)Affichage.o -c $(SRC)Affichage.c $(CFLAGS)

Donnees.o: $(SRC)Donnees.c
	$(COMPIL) -o $(OBJ)Donnees.o -c $(SRC)Donnees.c $(CFLAGS) $(PARA)

Median.o: $(SRC)Median.c $(INC)Pile.h
	$(COMPIL) -o $(OBJ)Median.o -c $(SRC)Median.c $(CFLAGS) $(PARA)

Gaussien.o: $(SRC)Gaussien.c
	$(COMPIL) -o $(OBJ)Gaussien.o -c $(SRC)Gaussien.c $(CFLAGS) $(PARA)

Marching.o: $(SRC)Marching.c
	$(COMPIL) -o $(OBJ)Marching.o -c $(SRC)Marching.c $(CFLAGS) $(PARA)

main.o: $(SRC)main.c $(INC)Pile.h $(INC)Affichage.h $(INC)Median.h $(INC)Gaussien.h $(INC)Donnees.h $(INC)Marching.h $(INC)Affichage3D.h $(INC)Types.h $(INC)dbh.h
	$(COMPIL) -o $(OBJ)main.o -c $(SRC)main.c $(CFLAGS)

clean:
	rm -rf $(OBJ)*.o
	rm bin/Marching
```

Figure 3. Structure du Makefile

3.2.2 Extraction d'iso-surface

Comme dit précédemment, l'algorithme du Marching cubes fût le plus complexe à comprendre, dans le sens où il se sert de concepts de plus haut niveau que le filtre médian et le filtre gaussien qui ne sont que de la manipulation de tableaux. Ainsi, j'ai passé plus de temps à essayer de comprendre son fonctionnement que pour les algorithmes de filtres. Cependant, après avoir cherché de la documentation sur internet [7] [8] [9] [10] et avoir revu le C en détails grâce aux précédentes implémentations, j'ai pu sans trop de difficultés passer à celle du Marching cubes.

Ainsi, de la même manière que précédemment, j'ai crée un fichier appelé Marching.c, qui possédait une seule méthode utile, capable de lancer l'algorithme du Marching cubes en fonction de certains paramètres spécifiques comme l'isovaleur. Le résultat du Marching cubes étant un ensemble de triangles, il me fallait ensuite récupérer cet ensemble de triangles afin que le fichier en charge de l'affichage OpenGL puisse les récupérer. Pour cela, j'ai donc créer une structure nommée triangle2D, qui contient trois valeurs qui ne sont autres que les trois points d'un triangle représenté en deux dimensions, et ai ensuite déclarer un pointeur vers cette structure. Ainsi, au fur et à mesure

que l'algorithme progresse dans sa recherche de triangles, il stocke les triangles résultats dans ce pointeur. Ce dernier est donc visible par tous les autres fichiers, et notamment celui qui gère l'affichage 3D grâce à OpenGL. A l'intérieur de ce fichier on n'a donc plus qu'à lire l'ensemble des triangles qui composent le résultat du Marching cubes, et à tracer les triangles adéquats pour former l'image complète grâce aux points qui composent les triangles.

Une fois implémenté, on peut appliquer l'algorithme sur l'image virtuelle. Pour rappel, elle est de la forme $f(x, y, z) = \frac{x^2}{a} + \frac{y^2}{b} + \frac{z^2}{c}$ où a, b et c sont des constantes.

Ainsi, si on considère $a = 2$, $b = 2$ et $c = 2$, on a alors la représentation d'une sphère grâce à l'algorithme du Marching Cubes.

Voici un screenshot du résultat obtenu:

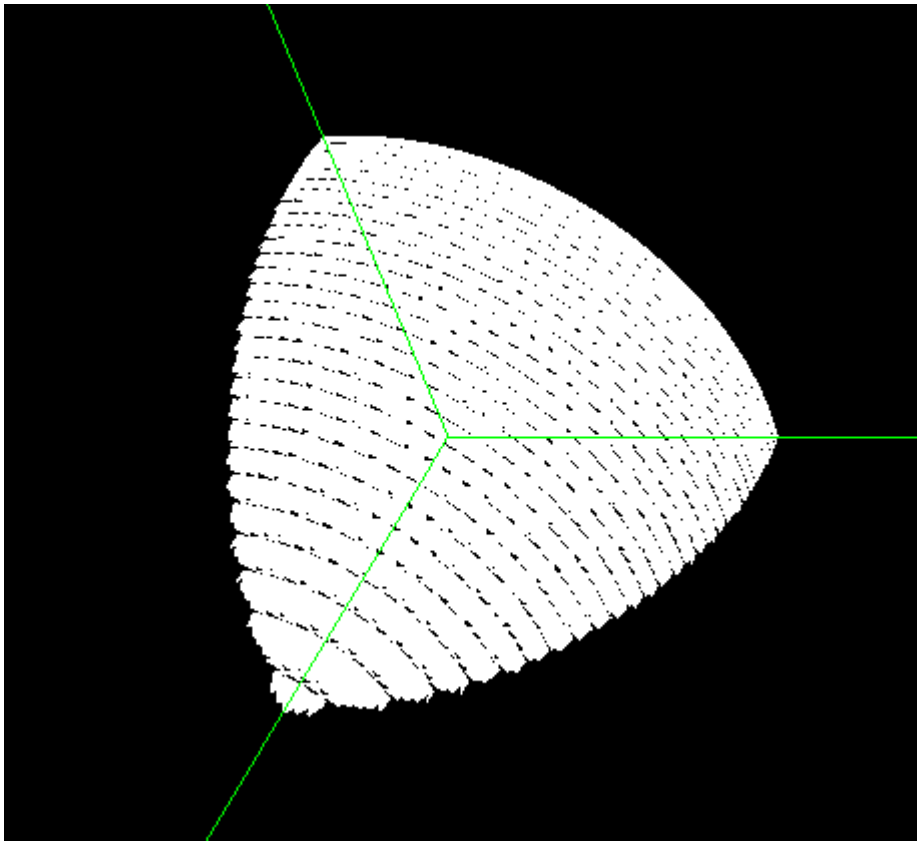


Figure 4. Résultat du Marching cubes sur une image virtuelle

Ici, l'image est remplie uniquement pour $x > 0$, $y > 0$ et $z > 0$. On a donc une représentation partielle de la sphère.

Lors de l'affichage de la fenêtre OpenGL, il est alors possible de faire tourner l'image, la translater ou bien effectuer un zoom. Tout ceci permet donc de voir l'image sous tous les angles, et ainsi pouvoir l'observer selon le point de vue que l'on souhaite.

3.3 Parallélisation

Le but de la parallélisation est d'exploiter au mieux l'architecture du serveur dont on se sert afin de réduire le temps de calcul d'une application. Pour cela, il peut s'avérer utile de savoir à l'intérieur du code de l'application quelles sont les fonctions qui demandent le plus de ressources, afin de paralléliser celles qui en utilisent le plus.

Dans cette optique, il existe un outils très utile, dont le nom est gprof, qui permet de faire du profiling de code, et ainsi savoir exactement le temps passé à l'intérieur de chaque fonction du code après compilation et exécution. Ainsi, dans un premier temps, je me suis servis de l'outil gprof pour savoir exactement où placer mes directives OpenMP afin d'optimiser leur utilisation.

J'ai ensuite assez rapidement compris comment utiliser ces directives OpenMP, et les ai donc introduites au bon endroit dans mon code afin d'être le plus efficace possible. Cela a eu un effet immédiat sur le temps de calcul, que je mesurais grâce à la commande Linux « time ». Ainsi, j'ai pu constater que celui-ci était réduit par 5 ou 6 selon que j'utilisais ou pas les directives OpenMP (voir section **5.1.5** pour le Benchmark).

De la même façon que pour les algorithmes de traitement d'image, j'ai commencé à insérer mes directives OpenMP sur le code du filtre médian, car il était le plus simple. J'ai ensuite progressé en complexité en insérant mes directives respectivement dans le code du filtre gaussien puis dans celui du Marching cubes. Ce dernier fût le plus difficile à paralléliser, car il nécessitait la pose d'un verrou sur certaines variables partagées, afin que les différentes tâches lancées n'écrasent pas le travail effectué par une autre tâche (voir pseudo code, profiling et pseudo code parallélisé du Marching cubes ci après). En effet, lors de l'accès en écriture à une variable partagée, c'est un problème classique que de ne donner la main qu'à une seule tâche à la fois. Mais dans l'ensemble, la parallélisation ne fût pas extrêmement compliquée dans le sens où les cours que j'avais récupérés sur internet expliquait très bien ce principe [11] [12] [13].

Ensuite, pour tester les résultats des algorithmes après les avoir parallélisés, j'ai comparé ces derniers aux résultats obtenus en appliquant le même algorithme, mais exécuté en séquentiel, sur une image identique. En comparant, par exemple, les intensités en chaque voxel après avoir appliqué un filtre, j'ai pu constater que les valeurs étaient toutes identiques. J'étais donc certain d'avoir un résultat cohérent, et comme le temps de calcul était largement réduit, la parallélisation se déroulait correctement. Pour le Marchin Cubes, le test a été beaucoup plus rapide dans le sens où le résultat à vérifier était une image en trois dimensions, beaucoup plus facile à comparer qu'une série de nombres.

Enfin, pour des raisons d'installation des librairies sur les blades du HPC, j'ai du changer de compilateur. Je suis donc passé de gcc à icc, le compilateur d'intel qui disposait déjà de la librairie OpenMP.

Les étapes de la parallélisation peuvent donc se résumer ainsi:

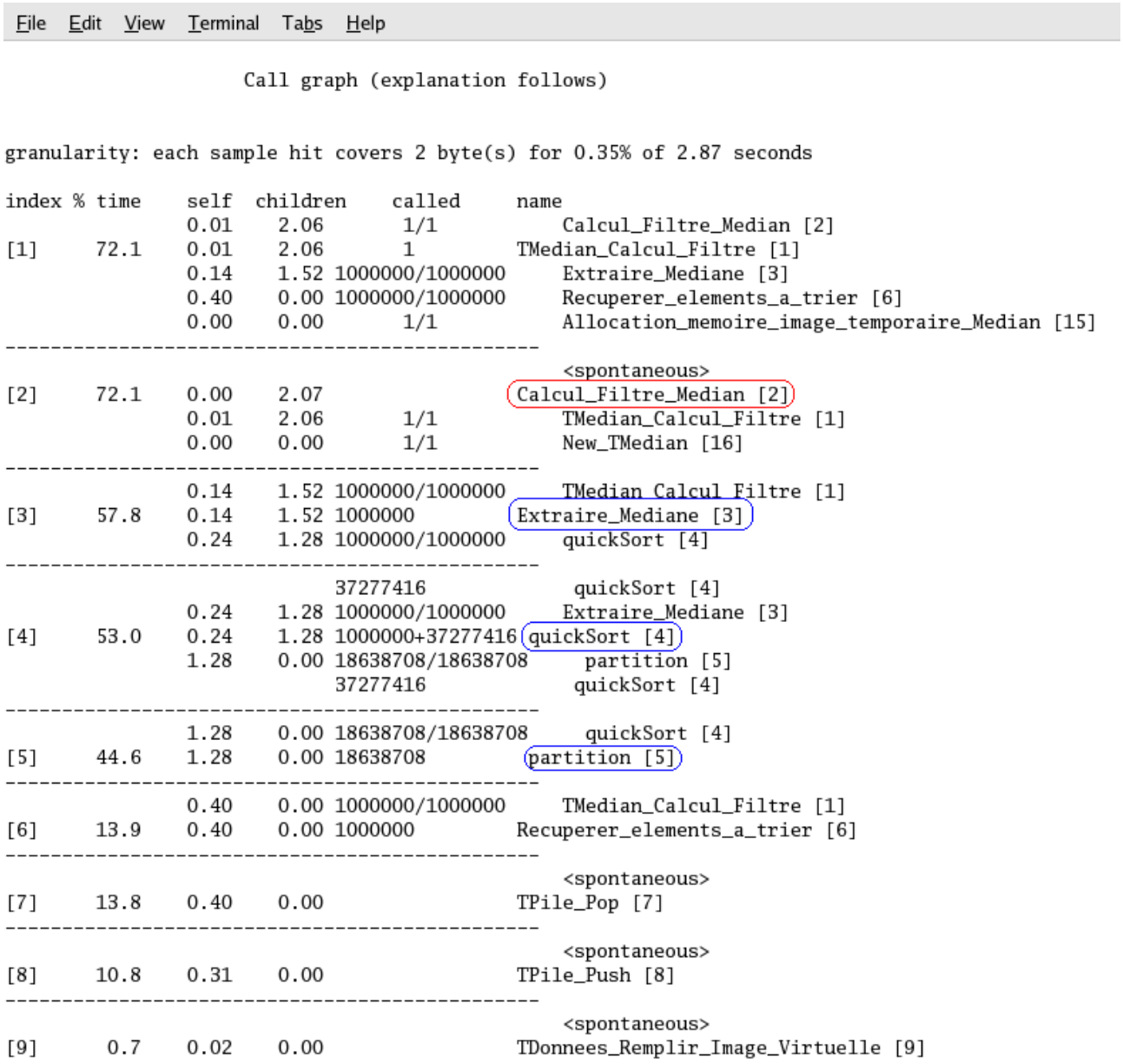
- Profiling du code avec Gprof
- Insertion de directives OpenMP
- Compiler le code
- Déclarer le nombre de tâches à lacer via la commande Linux « export OMP_NUM_THREADS = (le nombre de tâches souhaitées) »
- Tests sur le bon fonctionnement des algorithmes après insertion des directives

Afin de mieux expliciter ces phases, nous allons maintenant voir deux exemples d'applications. Pour cela, considérons l'exécution de l'algorithme du filtre médian et du Marching cubes sur une image de 1.000.000 voxels (100 x 100 x 100). On peut représenter le filtre médian par le pseudo-code suivant:

```
1  pour i de 0 à Taille_Image_En_X faire
2      pour j de 0 à Taille_Image_En_Y faire
3          pour k de 0 à Taille_Image_En_Z faire
4              Elements_Du_Filtre = Recuperer_elements_a_trier(i,j,k);
5              Image_intermediaire_Median[i][j][k] = Extraire_Mediane(Elements_Du_Filtre);
6          fin pour;
7      fin pour;
8  fin pour;
```

Figure 5. Pseudo code du filtre médian

Ensuite, si on utilise gprof avec ce code sur une image de 100 x 100 x 100, on obtient un profiling du filtre médian qui ressemble à ceci:



```
File  Edit  View  Terminal  Tabs  Help

                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.35% of 2.87 seconds

index % time    self  children    called     name
                0.01    2.06       1/1           Calcul_Filtre_Median [2]
[1]     72.1    0.01    2.06       1         TMedian_Calcul_Filtre [1]
                0.14    1.52 1000000/1000000     Extraire_Mediane [3]
                0.40    0.00 1000000/1000000     Recuperer_elements_a_trier [6]
                0.00    0.00       1/1           Allocation_memoire_image_temporaire_Median [15]
-----------------------------------------------
                                                 <spontaneous>
[2]     72.1    0.00    2.07                 Calcul_Filtre_Median [2]
                0.01    2.06       1/1           TMedian_Calcul_Filtre [1]
                0.00    0.00       1/1           New_TMedian [16]
-----------------------------------------------
                0.14    1.52 1000000/1000000     TMedian_Calcul_Filtre [1]
[3]     57.8    0.14    1.52 1000000         Extraire_Mediane [3]
                0.24    1.28 1000000/1000000     quickSort [4]
-----------------------------------------------
                             37277416             quickSort [4]
                0.24    1.28 1000000/1000000     Extraire_Mediane [3]
[4]     53.0    0.24    1.28 1000000+37277416 quickSort [4]
                1.28    0.00 18638708/18638708     partition [5]
                             37277416             quickSort [4]
-----------------------------------------------
                1.28    0.00 18638708/18638708     quickSort [4]
[5]     44.6    1.28    0.00 18638708          partition [5]
-----------------------------------------------
                0.40    0.00 1000000/1000000     TMedian_Calcul_Filtre [1]
[6]     13.9    0.40    0.00 1000000         Recuperer_elements_a_trier [6]
-----------------------------------------------
                                                 <spontaneous>
[7]     13.8    0.40    0.00                 TPile_Pop [7]
-----------------------------------------------
                                                 <spontaneous>
[8]     10.8    0.31    0.00                 TPile_Push [8]
-----------------------------------------------
                                                 <spontaneous>
[9]      0.7    0.02    0.00                 TDonnees_Remplir_Image_Virtuelle [9]
```

Figure 6. Profiling du code du filtre médian

Ici, on voit clairement que, lors de l'appel à la méthode Calcul_Filtre_Median (en rouge) les fonctions appelées par la suite et qui demandent le plus de ressources sont Extraire_Mediane, quicksort et partition (en bleu). Les deux dernières précitées étant des fonction utilisées pour le tri afin d'extraire la médiane de l'ensemble des valeurs contenues dans le filtre. Ainsi, afin de paralléliser le code, j'ai donc choisi de répartir le travail entre les différents processeurs lors de l'appel à la fonction Calcul_Filtre_Median. Pour cela, en fonction de l'indice de la première boucle pour (cf. **Figure 5.**), c'est le processeur dont le numéro est égal à l'indice modulo le nombre de processeurs qui va effectuer le travail. Voici comment cela on peut représenter cela en pseudo-code:

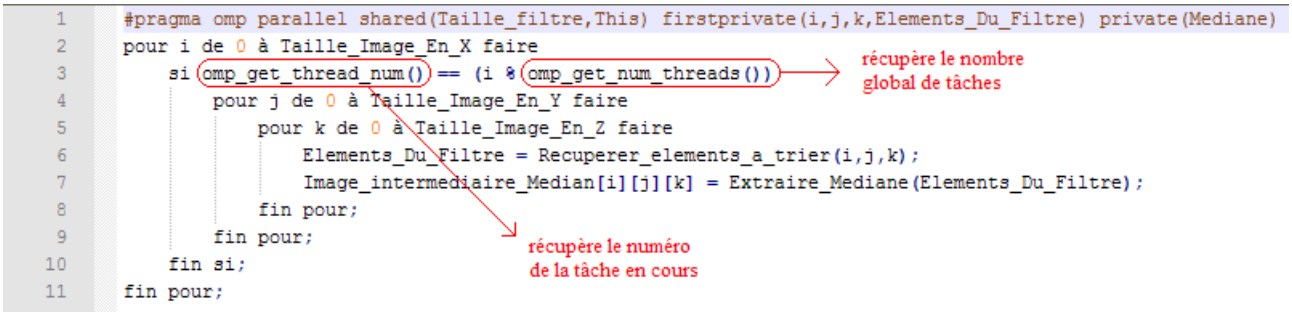


Figure 7. Pseudo code du filtre médian avec directives OpenMP

Ainsi, lorsque le système rentre dans la première boucle pour, il répartit le travail en fonction du nombre de tâches que l'on a lancé. On peut représenter la répartition des rôles de la manière suivante:

boucle avec i = 0 → processeur numéro 0
boucle avec i = 1 → processeur numéro 1
⋮
boucle avec i = (nombre de processeurs – 1) → processeur numéro (nombre de processeurs -1)
boucle avec i = nombre de processeurs → processeur numéro 0
boucle avec i = (nombre de processeurs + 1) → processeur numéro 1
etc...

Ainsi, la répartition des tâches est équitable et chaque processeur effectue globalement la même quantité de travail. Avec ce type d'implémentation (via OpenMP), il n'est pas possible de faire de la parallélisation multi-blade. Pour cela, il faudrait passer une implémentation via MPI afin de faire communiquer plusieurs blades entre elles et ainsi répartir le travail entre davantage de processeurs. Par manque de temps, lors de ce stage, je ne me suis uniquement attelé à de la parallélisation multiprocesseurs sur une seule blade via OpenMP. Cependant, les routines OpenMP et MPI sont indépendantes, ce qui éviterai d'avoir à réécrire les routines OpenMP si l'on décide de rajouter de la parallélisation multi-blade via MPI.

Si l'on considère maintenant l'algorithme du Marching cubes, comme dit précédemment il a été plus difficile à paralléliser car il nécessitait l'ajout d'un verrou afin de protéger l'accès en écriture aux variables partagées. Dans notre cas, c'est le pointeur contenant l'ensemble des triangles résultant du Marching cubes qui a dû être protégé (fonction Ajoute_triangles ci-dessous)

Voici ce que donne l'algorithme du Marching cubes écrit en pseudo code:

```
pour i de 0 à Taille_Image_En_X faire
    pour j de 0 à Taille_Image_En_Y faire
        pour k de 0 à Taille_Image_En_Z faire
            NombreTrianglesEnCours = Calcule_Triangles_Dun_Cube(triangles_resultats,i,j,k);
            si (NombreTrianglesEnCours != 0)
                Ajoute_triangles(Ensemble_des_triangles,triangles_resultats);
        fin pour;
    fin pour;
fin pour;
```

Figure 8. Pseudo code du Marching cubes

Ensuite, si l'on utilise gprof de la même manière que décrite pour le filtre médian précédemment (image de 100 x 100 x 100), on obtient un profiling du code qui ressemble alors à ceci:

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 9.99% of 0.10 seconds

index % time    self  children    called     name
                0.00    0.08       1/1          Marching_Cubes [2]
[1]     80.0    0.00    0.08       1          TMarching_Calculer_Isosurface [1]
                0.06    0.01  970299/970299      Calcule_Triangles_Dun_Cube [3]
                0.01    0.00    1714/1714        Ajoute_triangles [6]
-----------------------------------------------
                                                 <spontaneous>
[2]     80.0    0.00    0.08                   Marching_Cubes [2]
                0.00    0.08       1/1          TMarching_Calculer_Isosurface [1]
                0.00    0.00       1/1          New_TMarching [8]
-----------------------------------------------
                0.06    0.01  970299/970299      TMarching_Calculer_Isosurface [1]
[3]     70.0    0.06    0.01  970299           Calcule_Triangles_Dun_Cube [3]
                0.01    0.00  970299/970299      Calcule_Un_Triangle [5]
-----------------------------------------------
                                                 <spontaneous>
[4]     20.0    0.02    0.00                   TDonnees_Remplir_Image_Virtuelle [4]
-----------------------------------------------
                0.01    0.00  970299/970299      Calcule_Triangles_Dun_Cube [3]
[5]     10.0    0.01    0.00  970299           Calcule_Un_Triangle [5]
                0.00    0.00    7454/7454        VertexInterp [7]
-----------------------------------------------
                0.01    0.00    1714/1714        TMarching_Calculer_Isosurface [1]
[6]     10.0    0.01    0.00    1714           Ajoute_triangles [6]
-----------------------------------------------
                0.00    0.00    7454/7454        Calcule_Un_Triangle [5]
[7]      0.0    0.00    0.00    7454           VertexInterp [7]
-----------------------------------------------
                0.00    0.00       1/1          Marching_Cubes [2]
[8]      0.0    0.00    0.00       1          New_TMarching [8]
                0.00    0.00       1/1          TMarching_Init [9]
-----------------------------------------------
                0.00    0.00       1/1          New_TMarching [8]
[9]      0.0    0.00    0.00       1          TMarching_Init [9]
-----------------------------------------------
```

Figure 9. Profiling du code du Marching cubes

Ainsi, il apparaît clairement que, après l'appel dans le fichier principal à la fonction Marching_cubes (en rouge), la fonction appelées par la suite qui demande le plus de ressources est de loin Calcule_Triangles_Dun_Cube (en bleu). Ainsi, c'est ici qu'il faut placer les directives OpenMP afin d'être le plus efficace possible. Le résultat obtenu après parallélisation est explicité par le pseudo code suivant:

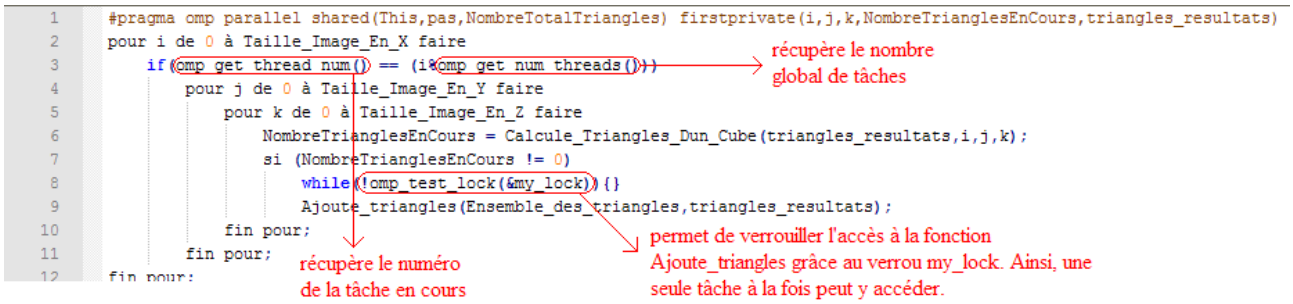


Figure 10. Pseudo code du Marching cubes parallélisé

Le verrou permet donc l'accès unique à une tâche, ce qui évite l'écrasement par d'autres tâches des variables manipulées et sécurise ainsi les données.

3.4 Intégration HPC / Visualisation

Ne possédant pas de carte graphique, les blades du HPC ne pouvaient donc pas servir pour transformer le buffer graphique calculé via OpenGL en l'image elle même. En effet, lors de l'affichage en OpenGL de l'image traitée, c'est alors la carte graphique de mon ordinateur personnel (ThinkPad) qui était utilisée. Ainsi, l'image mettait presque deux heures à s'afficher, car cette dernière n'est pas la plus récente des cartes graphiques puisque c'est une ATI mobility Radeon X300.

Voici alors comment aurait pu être représentée l'architecture utilisée:

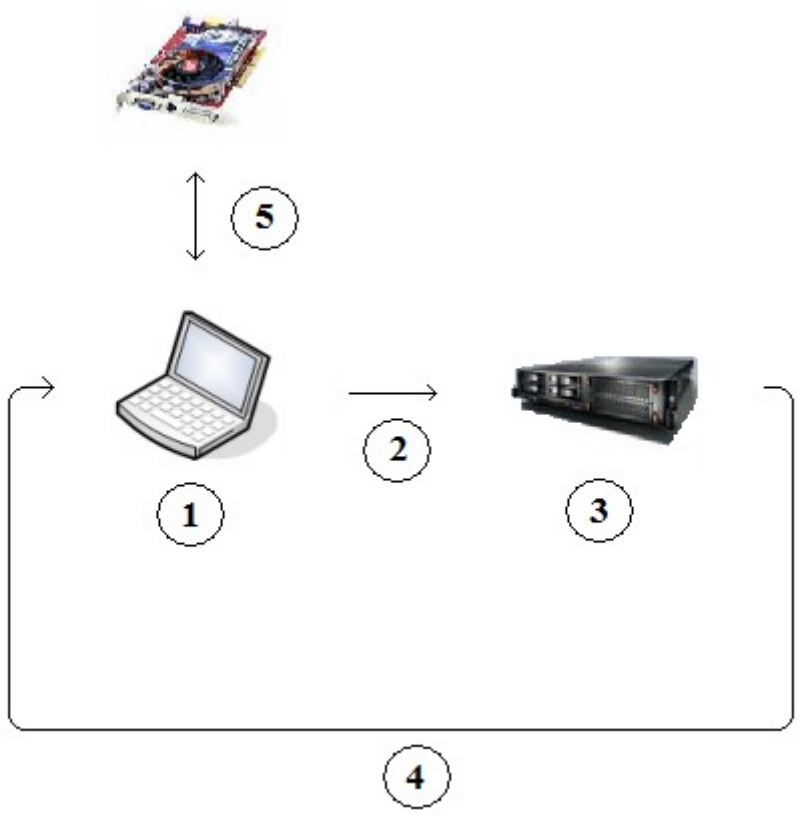


Figure 11. Schéma de l'architecture utilisée lors de l'utilisation d'une blade sans carte graphique

Les différentes étapes peuvent donc se résumer ainsi:

(1) Connexion à la blade par le protocole ssh via un shell de type Xshell ou putty

(2) Ordre de lancement de l'application

(3) Exécution de l'application

(4) Envoi du buffer graphique au client (cf **Figure 23.**)

(5) Affichage de l'image par la carte graphique du ThinkPad (ATI mobility Radeon X300)

Afin de faire communiquer la blade effectuant les calculs aux serveurs de la salle immersive gérants l'affichage sur les 4 projecteurs, il fallait en plus relier la blade où les calculs étaient effectués au réseau interne. En effet, si cela n'est pas le cas, les serveurs de la salle immersive auraient bien été capable d'envoyer des messages à la blade, mais la réciproque n'aurait pas été possible.

Pour ces raisons, et parce que les blades du HPC sont souvent utilisées par des clients d'IBM partout dans le monde et/ou par le personnel d'IBM, il devenait évident qu'il nous fallait réserver une blade pour mon travail personnel, sur laquelle on pourrait monter une carte graphique et la relier au réseau interne sans risques d'empiéter sur les travaux de tierces personnes. Ainsi, nous avons donc réserver une blade de type HS21 XM où nous avons monté une carte graphique NVIDIA quadro 3007.

Il a alors fallut installer divers éléments:

- Un OS, en l'occurrence RedHat 4, qui est une distribution de Linux
- Les drivers nécessaires au bon fonctionnement des périphériques, notamment ceux de la carte graphique
- Un serveur VNC (VNC Server 4) afin d'assurer la communication ThinkPad – Blade via VNC
- Toutes les librairies dont je me servais jusqu'à présent sur les blades du HPC

Afin de pouvoir se connecter en remote (à distance) à cette lame, nous avons installé sur mon ThinkPad (qui est donc le client) VNC Viewer 4, et avons dû créer toutes les autorisations d'accès sur les serveurs gérants les quatre écrans de la salle immersive.

VNC se compose de deux parties, le client et le serveur. Le serveur est le programme esclave sur la machine qui partage son écran, et le client (appelé aussi le "viewer") est le programme maître qui regarde et interagit éventuellement avec le serveur. Ainsi, lorsque l'on passe par ce logiciel, l'affichage est totalement déporté sur le serveur, qui doit alors posséder sa propre carte graphique. Ceci permet d'afficher beaucoup plus rapidement l'image, car la carte graphique installée est généralement beaucoup plus puissante sur le serveur que sur le client.

Voici alors comment on pourrait représenter l'architecture utilisée:

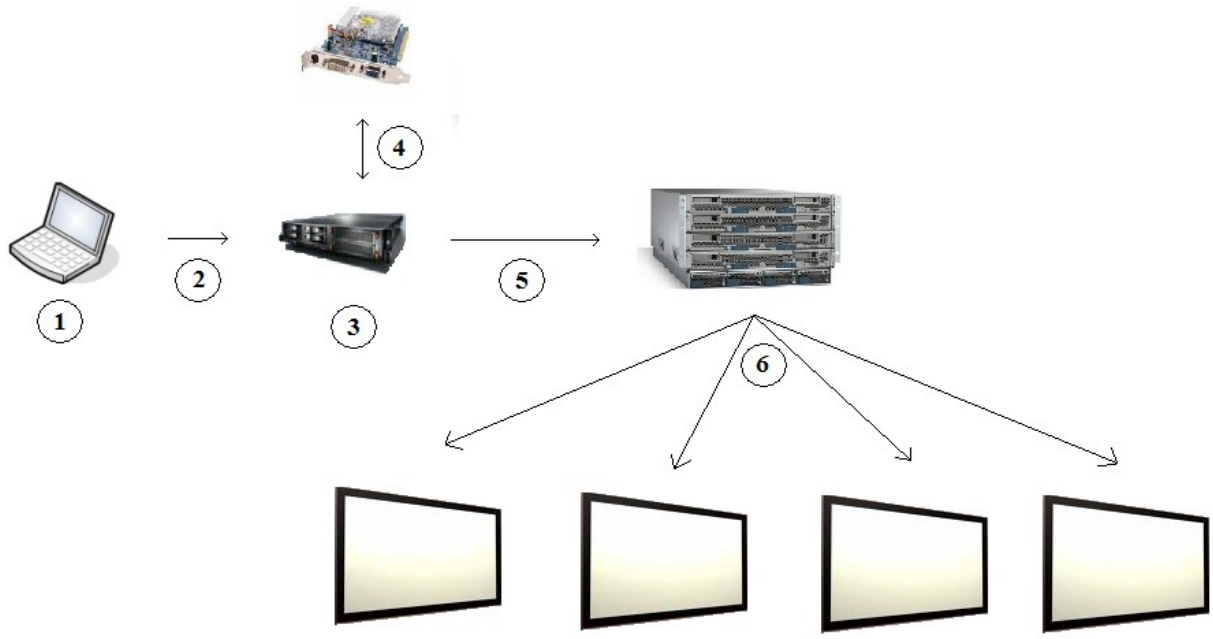


Figure 12. Schéma de l'architecture lors de l'affichage en salle immersive en passant par une blade avec carte graphique

Les différentes étapes peuvent donc se résumer ainsi:

(1) Lancement du client VNC (VNC Viewer 4) sur le ThinkPad

(2) Connexion à la blade

(3) Exécution de l'application

(4) Calcul du buffer graphique puis représentation de l'image

(5) Connexion puis envoi de l'image aux quatre serveurs de la salle immersive gérants chacun un écran.

(6) Répartition du flux vidéo entre les quatre projecteurs en fonction du fichier de configuration wall.cfg

Les étapes pour afficher dans la salle immersive sont donc les suivantes:

- Démarrer les quatre projecteurs
- Lancer VNC Viewer sur le ThinkPad et se connecter à la blade
- Démarrer svn (via svn_enable) avec la bonne configuration, c'est à dire le bon fichier wall.cfg qui regroupe l'ensemble des informations nécessaire au découpage de l'image sur les quatre projecteurs
- Se connecter aux serveurs gérants les quatre projecteurs et qui ont pour nom dv03, dv07, dv08 et dv09 grâce au protocole ssh
- Lancer, à partir du client (ThinkPad) et via VNC Viewer, l'application utilisant OpenGL

La commande rvn_dashboard permettant, quand à elle, d'accéder à d'autres paramètres très utiles comme la qualité de l'image affichée.

4. Conclusion

4.1 Résultats obtenus

4.1.1 Benchmark

Un des buts principaux de ce stage était de se rapprocher du temps réel lors de l'exécution de l'application. J'ai donc parallélisé les algorithmes, ce qui a entraîné une réduction du temps de calcul importante car le serveur dont je disposais était composé de huit processeurs. Afin de mettre en évidence la réduction du temps de calcul lors de l'exécution des différents algorithmes après parallélisation, j'ai donc été amené à réaliser un Benchmark, qui présente une comparaison entre le temps d'exécution avant et après parallélisation en fonction de la taille de l'image. Les détails de ce Benchmark sont disponibles dans la section **5.1.4** Benchmark.

Dans celui-ci, la première chose dont on se rend compte, c'est que la réduction du temps de calcul n'est pas polynomiale (voir ci dessous, Marching cubes pour les explications). En effet, je disposais d'un serveur avec huit processeurs, mais le rapport du temps d'exécution entre séquentiel et parallélisé est en moyenne aux alentours de 7. Ceci étant, la réduction du temps de calcul est conséquente, et permet de s'approcher du temps réel que l'on pourrait aisément réaliser avec deux ou trois blades de type LS22.

Intégration des données

Pour effectuer les traitements sur l'image, il faut tout d'abord remplir le tableau à 3 dimensions qui correspond à l'intensité en chaque voxel de l'image. Ainsi, plus l'image est grande, plus le tableau sera rempli lentement, il est donc intéressant de savoir quel temps va mettre l'application à charger les données. Cette section n'est pas réellement indispensable mais permet tout de même de se faire une idée quand au temps qu'il faut pour remplir correctement le tableau à 3 dimensions qui contient l'image.

Bien évidemment, les données du bâton percé sont beaucoup plus lentes à intégrer, car cela passe par une lecture de fichiers et une conversion des valeurs obtenues (codées sur deux octets) afin d'obtenir l'intensité exacte en chaque voxel. Ceci explique donc l'avant dernière ligne du résultat présenté dans la **Table 1**.

Ainsi, pour intégrer les données, on peut aisément constater que le temps de calcul n'est pas très important, mais prend tout de même cinq minutes en séquentiel en ce qui concerne le bâton percé. J'ai pu réduire celui-ci à une minute, une intégration des données en temps réel étant donc largement envisageable avec deux ou trois blades de type LS22, et avec insertion de directives MPI afin de les faire communiquer pour se partager le travail.

Filtre médian

Dû à la fonction de tri qu'il utilise pour extraire la médiane d'un ensemble de valeurs, le filtre médian est l'algorithme implémenté lors de ce stage qui consomme le plus de ressources en terme de CPU. Ainsi, appliqué sur les données du bâton percé, composé approximativement de 2.100.000.000 voxels, le filtre médian met environ 90 minutes à s'exécuter, alors que lorsqu'on introduit des directives OpenMP, cela permet de réduire le temps de calcul à 15 minutes. Le temps de calcul pour l'algorithme du filtre médian est donc divisé par 7, et cela conforte l'idée qu'avec deux ou trois blades de type LS22 on puisse exécuter l'application en temps réel sur un jeu de données comprenant environ 2.100.000.000 voxels.

Dans l'ensemble, grâce à l'insertion de directives OpenMP afin de paralléliser le code du filtre médian sur les serveurs multiprocesseurs, il est donc possible de réduire le temps de calcul entre 6 et 7 fois. Tout ceci en lançant 8 tâches différentes (le maximum sur une seul nœud de type LS22 pour apporter un réel gain de temps, car les blades de type LS22 disposent de huit processeurs), ce qui nous donne donc des résultats satisfaisants car les routines OpenMP sont indépendantes des routines MPI (dont on se sert pour faire communiquer deux nœuds afin de lancer encore plus de threads. Il est alors possible de lancer 16 threads pour 2 nœuds, 24 pour 3 nœuds, etc.).

Filtre gaussien

Si l'on se fie à la complexité des trois algorithmes importants rentrants dans le cadre de ce stage (Filtre médian, gaussien et Marching cubes), le filtre gaussien est celui qui se situe dans la moyenne des deux autres en terme de temps d'exécution pour une même image. Cela se vérifie par les temps obtenus lors des mesures qui sont explicitées dans la **Table 3.**

En ce qui concerne le filtre gaussien, on peut déjà constater que le temps mis lors de l'exécution est bien moins important que pour le filtre médian, qui reste l'algorithme le plus gourmand en ressources. Ainsi, sur les données du bâton percé, le filtre gaussien met environ trois minutes à s'exécuter lorsqu'il est parallélisé, alors que le filtre médian en met quinze soit cinq fois plus de temps de calcul.

Ainsi, lorsqu'on regarde le Benchmark du le filtre gaussien, on s'aperçoit qu'il est possible de réduire le temps de calcul entre 8 et 9 fois, ce qui est appréciable. Celui-ci est beaucoup plus diminué que pour le filtre médian car dans le filtre médian toute la section de tri pour extraire la médiane n'est pas parallélisée. De plus, le filtre gaussien est un algorithme qui n'utilise que très peu de fonctions annexes non parallélisées: après avoir calculé le noyau gaussien, ce qui n'est effectué qu'une seule fois lors du lancement de l'algorithme, il suffit de récupérer les intensités des voxels voisins du voxel courant afin, ensuite, de mettre à jour le coefficient du voxel courant. Aucune structure de tri ou tout autre algorithme dont la consommation en temps de calcul est coûteuse n'est utilisé.

Marching cubes

Le Marching cubes étant le plus rapide des trois principaux algorithmes implémentés lors de ce stage, il n'est pas celui qui demande le plus de ressources mémoires. Ainsi, avec une seule blade de type LS22, l'exécution se fait en à peine une minute et demi pour les données du bâton percé quand l'algorithme est parallélisé, contre dix minutes lorsqu'il s'exécute en séquentiel. De fait, le temps de calcul est divisé par sept, ce qui est satisfaisant lorsque l'on sait que la blade LS22 dispose de huit processeurs.

Pour prendre les mesure du temps de calcul du Marching cubes, et afin d'obtenir un résultat cohérent en terme de quantité de travail effectué par l'algorithme, le nombre de triangles est augmenté à chaque changement de taille d'image. Les mesures étant principalement effectuées sur une image virtuelle, l'isovaleur entrant en paramètre dans l'algorithme est donc elle aussi incrémentée. Le résultat est un nombre toujours plus grand de triangles obtenus en résultat du Marching cubes. Cependant, on peut constater que la réduction du temps de calcul entre l'algorithme exécuté en séquentiel et celui parallélisé varie entre cinq et sept environ. Cela vient du fait que le temps de calcul est dépendant de la quantité de sommets présents à l'intérieur de la surface représentée par l'iso-surface. Ainsi, plus le nombre de sommets présents à l'intérieur de l'iso-surface sera grand, moins le temps de calcul sera réduit car le traitement effectué en chaque cube n'est pas parallélisé, il est exécuté en séquentiel. Ainsi, comme explicité pour le filtre médian dans la section **3.3 Parallélisation**, la répartition du travail s'effectue en chaque cube, c'est à dire que le travail effectué à l'intérieur d'un cube ne sera effectué que par une seule tâche mais qu'une tâche ne traitera pas l'ensemble des cubes. De fait, si le nombre de cubes présents à l'intérieur de l'iso-surface est important, la réduction du temps de calcul en sera d'autant diminuée.

Cela vient du fait que l'accélération, autrement dit le gain en performance d'un code parallèle, est dépendant du temps relatif à la partie séquentielle, autrement dit les régions du code qui ne sont pas parallélisées (le traitement à l'intérieur de chaque cube en l'occurrence). Ainsi, si on note t_s le temps relatif à la partie séquentielle, i.e. le temps de calcul à l'intérieur d'une région séquentielle, t_p le temps relatif à la région parallèle et N_t le nombre de tâches lancées, on peut alors majorer l'accélération $S(N_t)$ par la fraction séquentielle $\frac{1}{t_s}$ du programme. Cela découle de la loi dite de « AMDHAL » qui est définit comme suit:

$$S(N_t) = \frac{1}{t_s + \frac{t_p}{N_t}}$$

Ainsi, on peut représenter l'évolution du gain en vitesse d'exécution d'un programme en fonction du nombre de processeurs par le graphique suivant:

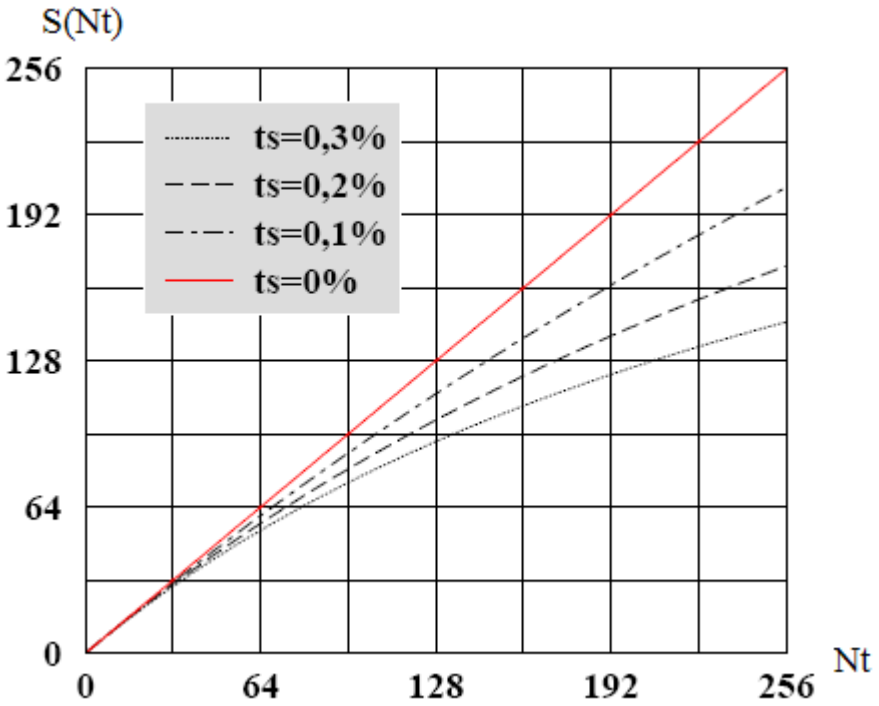


Figure 13. Évolution du gain en vitesse d'exécution d'un programme en fonction du nombre de processeurs

De plus, pour le bâton percé, le nombre de triangles à afficher avec OpenGL est approximativement de 38.000.000 lorsqu'on utilise une iso-valeur de 1200, ce qui n'est pas très volumineux comparé au nombre de triangles résultants du Marching cubes sur l'image virtuelle. La quantité de triangles à afficher est un paramètre à prendre en compte lors de l'affichage, car c'est ce dernier qui va déterminer le temps que mettra la carte graphique à afficher la fenêtre OpenGL et à la rafraîchir.

4.1.2 Visualisation

Lors de mon premier test d'affichage du bâton percé, le code tournait encore sur les serveurs multiprocesseurs du HPC, l'affichage des quelques quarante millions de triangles qui composaient l'image était alors réalisé par la carte graphique de mon ThinkPad (cf **Figure 11**). Ainsi, l'affichage de la fenêtre prenait plus de deux heures, ce qui est impensable quand on veut fait du temps réel. Ainsi, après réservation de la blade de type HS21 et après avoir installé la carte graphique NVIDIA quadro 3007, le temps d'affichage était beaucoup plus raisonnable car il ne dépasse jamais les vingts secondes. En contrepartie, le nombre de processeurs disponibles sur la blade de type HS21 est moins important que sur la blade de type LS22 et le temps d'exécution est donc bien plus lent.

J'ai donc affiché le résultat du Marching cubes sur le bâton percé afin d'en visualiser la forme, voici le résultat:

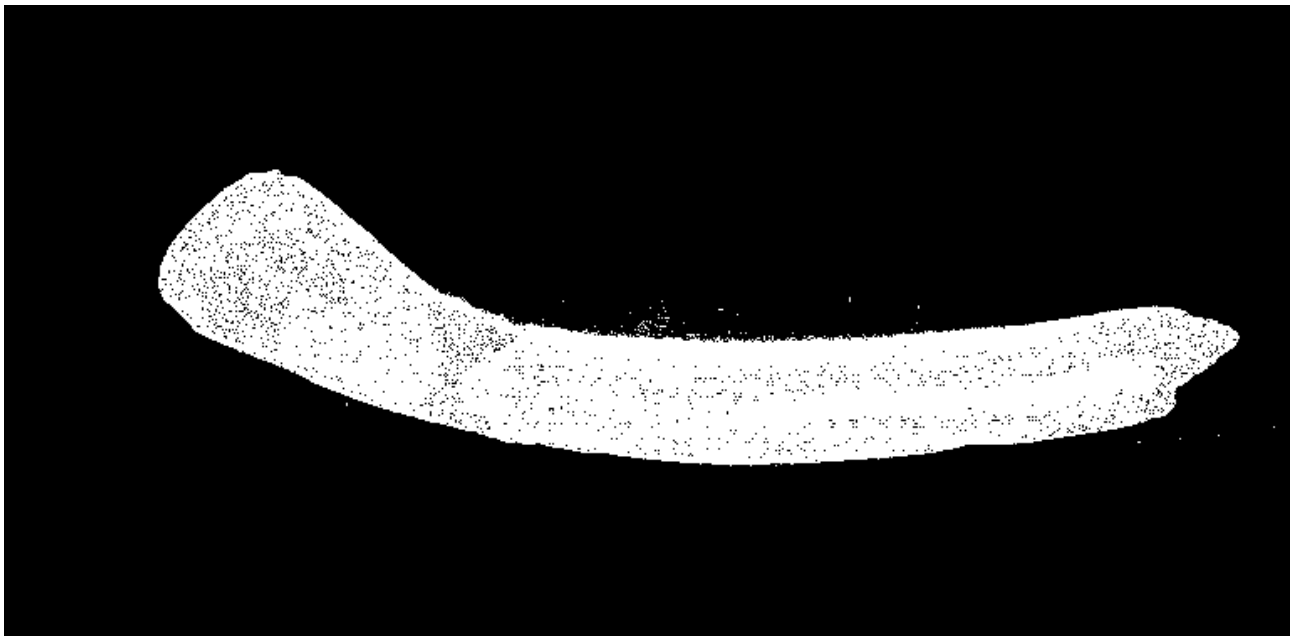


Figure 14. Affichage 3D du bâton percé

4.2 Difficultés rencontrées

Durant ce stage, il s'est posé à moi un certain nombre de difficultés plus ou moins faciles à résoudre. Ainsi, trois points techniques m'ont particulièrement posé problème, mais ont pu être résolus assez facilement grâce à l'intervention de personnes compétentes dans le domaine où j'éprouvais ces difficultés:

- L'installation de librairies sous Linux, notamment OpenGL sur les serveurs du HPC. Ce point a pu être résolu grâce à Xavier VASQUES, qui m'a donc bien aidé et grâce à qui j'ai pu me débrouiller seul ensuite quand il a fallut installer les librairies dont je me servais sur la blade que j'ai réservée afin de pouvoir afficher dans la salle immersive.

- L'intégration des données du bâton percé dans l'application. Ce point a été résolu grâce à Gérard SUBSOL, qui m'a donc bien aidé en me consacrant un peu de son précieux temps afin que je puisse lire un fichier de type ANALYSE et remplir le tableau correspondant à l'image 3D dans mon application. Ce point n'était pas évident dans le sens où je n'avais pas beaucoup de notions sur le codage en machine des nombres, ce qui n'est plus le cas et cela m'a donc permis d'enrichir mes compétences.

- La communication entre une blade et la salle immersive. Grâce à l'aide de François-René ROUGEAUX, j'ai pu comprendre comment faire communiquer une blade avec les serveurs de la salle immersive. Merci aussi à François-René pour m'avoir expliqué en détail les

étapes qu'il fallait réaliser, aussi bien matérielles (réservation lame, branchage en VLAN 4 sur réseau interne, montage d'une carte graphique, etc.) que logicielles (installation de VNC Viewer / Server et configuration, autorisations d'accès, etc.).

Sur un plan moins technique, j'ai éprouvé au début quelques difficultés d'adaptation au mode de travail chez IBM. En effet, au début de mon stage, je pensais que la présence sur le site de centaines d'ingénieurs reconnus me permettrais de pouvoir tirer bénéfice de leurs compétences afin d'enrichir les miennes et de me faire aider dans ma mission ici. Cependant, je me suis vite rendu compte qu'il n'était pas évident de trouver la bonne personne capable d'apporter son aide sur des questions techniques. Mes tuteurs au sein d'IBM m'ont bien aidés dans ce sens, et j'ai progressivement pris mes marques jusqu'à être parfaitement intégré dans cette grande entreprise qu'est IBM. L'ambiance y étant conviviale, cela m'a fortement aidé.
De plus, la découverte de l'environnement de travail n'a pas été chose aisée non plus, car il a fallut me familiariser avec le grand nombre d'outils mis à notre disposition. Cependant, je m'y suis fait en très peu de temps et cela m'a ensuite permis de disposer d'une large palette d'outils très utiles tels que les RedBook ou Lotus Notes.

4.3 Apports

Au delà de l'apprentissage du monde de l'entreprise, ce stage m'a énormément apporté tant sur le plan technique que sur le plan humain. En effet, grâce à ce stage, il m'a été permis d'étudier le domaine dans lequel je me suis spécialisé, à savoir le traitement d'images et la visualisation 3D. C'est une chance de pouvoir développer ses compétences dans une entreprise telle que IBM, car elle offre la possibilité de demander de l'aide aux nombreux spécialistes qui la compose.

Ainsi, grâce au personnel présent chez IBM et à mes tuteurs de l'université, j'ai pu accroitre mes compétences tant dans mon domaine d'études que dans d'autres secteurs que je connaissaient moins. La parallélisation est un bon exemple de ce que j'ai pu étudier lors de ce stage, car il est un domaine d'avenir et passionnant. De plus, j'ai pu apprendre beaucoup de choses sur l'architecture d'un ordinateur lors de l'utilisation des serveurs multiprocesseurs disponibles chez IBM car j'ai du avant tout comprendre ce que je manipulais afin de pouvoir m'en servir correctement.
De même, j'ai pu comprendre de façon un peu plus précise le fonctionnement d'un réseau dans une grande entreprise, car j'ai du en permanence jongler entre diverses machines afin d'y faire tourner mon application. Grâce à cela, j'ai pu aussi développer mon aptitude à utiliser Linux, car j'ai du plusieurs fois installer des librairies, changer certains paramètres du système ou encore comprendre le fonctionnement d'un Makefile et savoir en écrire un. Cela m'a permis de développer mon habilité à utiliser Linux, et j'ai pu découvrir de nombreuses fonctionnalités de ce dernier.
Un autre point que j'ai pris plaisir à découvrir est le codage en machine des nombres en fonction de leur type (entier signé / non signé, court / long, réel, double, etc.), car même si on ne se sert pas tous les jours de ce type de compétence, je le trouve extrêmement enrichissant au moins sur le plan de la culture personnelle.

Ainsi, je suis heureux d'avoir pu faire mon stage ici, car toutes ces choses que j'ai appris me seront bien utile pour la suite de ma carrière.

4.4 Perspectives

Lors de ce stage, j'ai réalisé tous les objectifs que je m'étais fixé. Cependant, avec un délais plus long, il aurait été possible d'améliorer encore plus l'application que j'ai développée. Ainsi, on pourrait par exemple

- Implémenter d'autres algorithmes de traitement d'images

- Passer à de la parallélisation multi-blades en utilisant donc la librairie MPI. Cela permettait de se rapprocher encore plus du temps réel. On peut donc imaginer utiliser deux ou trois blades de type LS22, insérer des directives MPI dans le code existant et ainsi pouvoir faire le traitement algorithmique en temps réel sur une image 3D d'environ 2 ou 3 milliards de voxels. On aurait lors une parallélisation efficace, car on exploiterai alors au maximum les ressources de chaque blade via OpenMP (les processeurs disponibles, tâche déjà réalisée) sur plusieurs blades via MPI (ce qui multiplierai donc le nombre de processeurs disponibles, tâche à réaliser).

- Développer un outil permettant l'affichage en salle immersive automatique, c'est à dire sans avoir à démarrer svn et se connecter aux serveurs gérants les quatre projecteurs à chaque fois

- Créer une interface graphique plus agréable, car pour l'instant elle est réduite au minimum

5. Annexes diverses

5.1 Description détaillée des concepts utilisés

5.1.1 Filtres graphiques

5.1.1.1 Qu'est ce qu'un filtre graphique?

Le filtrage consiste à appliquer une transformation (appelée *filtre*) à tout ou partie d'une image numérique en appliquant un opérateur. On distingue généralement les types de filtres suivants:

- Les *filtres passe-bas*, consistant à atténuer les composantes de l'image ayant une fréquence haute (pixels foncés). Ce type de filtrage est généralement utilisé pour atténuer le bruit de l'image, c'est la raison pour laquelle on parle habituellement de lissage. Les filtres moyenneurs sont un type de filtres passe-bas dont le principe est de faire la moyenne des valeurs des pixels avoisinants. Le résultat de ce filtre est une image plus floue.

- Les *filtres passe-haut*, à l'inverse des passe-bas, atténuent les composantes de basse fréquence de l'image et permettent notamment d'accentuer les détails et le contraste, c'est la raison pour laquelle le terme de "*filtre d'accentuation*" est parfois utilisé.

- Les *filtres passe-bande* permettant d'obtenir la différence entre l'image originale et celle obtenue par application d'un filtre passe-bas.

- Les *filtres directionnels* appliquant une transformation selon une direction donnée.

Un filtre est une transformation mathématique (appelée *produit de convolution*) permettant, pour chaque pixel de la zone à laquelle il s'applique, de modifier sa valeur en fonction des valeurs des pixels avoisinants, affectées de coefficients.

Le filtre est représenté par un tableau (matrice), caractérisé par ses dimensions et ses coefficients, dont le centre correspond au pixel concerné. Les coefficients du tableau déterminent les propriétés du filtre. Voici un exemple de filtre 3 x 3:

1	1	1
1	**3**	1
1	1	1

Ainsi le produit de la matrice image, généralement très grande car représentant l'image initiale (tableau de pixels) par le filtre donne une matrice correspondant à l'image traitée.

Dans ce stage, l'image à traiter n'est bien évidemment pas une image 2D en niveaux de gris mais une image 3D. Cela ne change pas grand chose hormis le fait que le filtre appliqué sera lui aussi en 3 dimensions.

La notion de bruit

Le bruit caractérise les parasites ou interférences d'un signal, c'est-à-dire les parties du signal déformées localement. Ainsi le bruit d'une image désigne les pixels de l'image dont l'intensité est très différente de celles des pixels voisins.

Le bruit peut provenir de différentes causes :

- Environnement lors de l'acquisition
- Qualité du capteur
- Qualité de l'échantillonnage

Grâce à un *lissage*, on peut réduire le bruit obtenu lors de l'acquisition des données. Cela modifie bien évidemment l'image originale et affecte donc la visualisation finale de cette dernière, mais si les propriétés du filtre sont bien choisies la qualité du rendu en sera d'autant améliorée.

5.1.1.2 Filtrage Médian

Applications

Dans certaines images, la valeur de certains pixels est beaucoup trop éloignée de ce qu'elle devrait être normalement. Par exemple, lorsqu'une image est composée de ce que l'on appelle le bruit *poivre et sel* (cas d'une image en noir et blanc ou certains pixels sont soit noir soit blanc), il peut être intéressant d'appliquer un filtre médian. De tels écarts peuvent s'expliquer, par exemple, par une mauvaise qualité des capteurs lors de l'acquisition des données de l'image. L'image sera alors moins jolie esthétiquement mais on pourra alors voir les détails de façon beaucoup plus visible.

Figure 15. Image avec un bruit poivre et sel affectant 25% des pixels

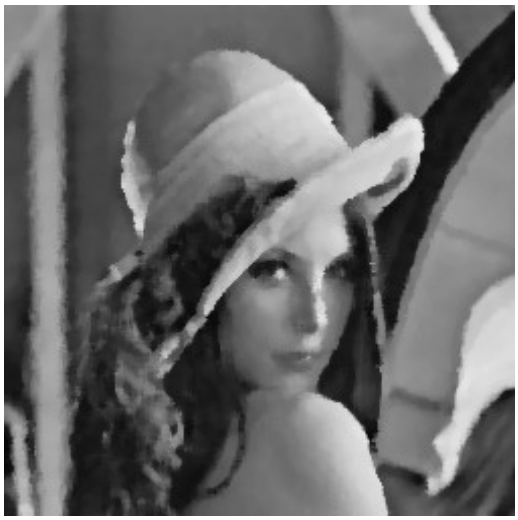


Figure 16. Filtrage médian 3 x 3 de l'image bruitée

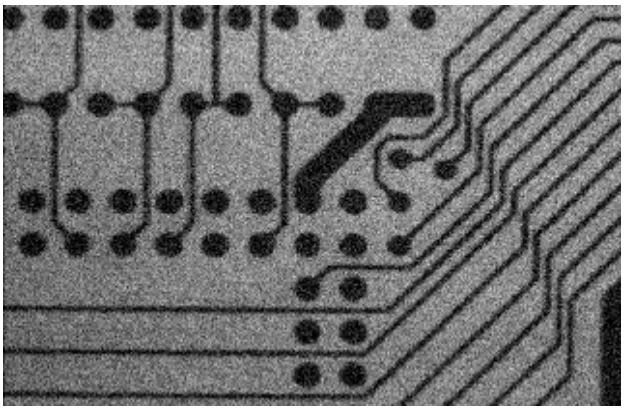


Figure 17. Image bruitée

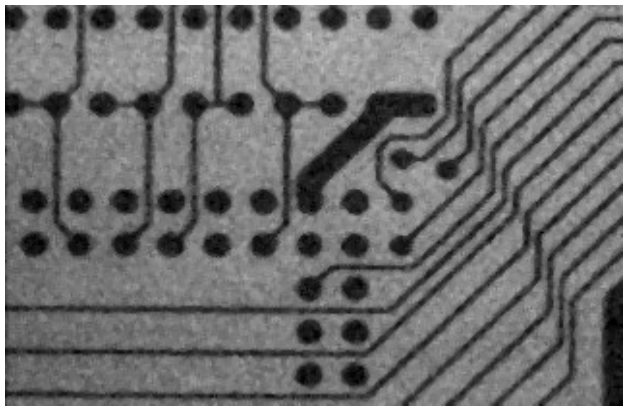


Figure 18. Application d'un filtre médian 3 x 3

L'exemple montre bien que le filtre médian respecte les contours, sans introduire de flou sensible, sauf si le bruit est vraiment trop important, auquel cas l'image qui résulte du filtre médian reste de bonne qualité si l'on tient compte de l'importance de la dégradation de l'image.

Principe

La médiane est une mesure statistique qui représente une alternative robuste à la moyenne. Considérons *n* valeurs numériques , on les ordonne de la plus petite à la plus grande. La médiane est alors la valeur placée au milieu de cette suite ordonnée.

Le principe du filtre médian est donc, comme son nom l'indique, de remplacer la valeur d'un pixel donné par un calcul de médiane grâce aux pixels voisins. Pour cela, il faut bien évidemment récupérer les valeurs des pixels voisins de celui que l'on est en train de traiter, en extraire la médiane qui correspondra à la nouvelle valeur de notre pixel. Toute l'efficacité d'un tel filtre dépend de l'algorithme de tri implémenté, dans le cadre de ce stage, deux tris différents ont été implémentés:

- Tri par bulle
- Tri « Quicksort », qui sera toujours utilisé car bien plus rapide que le tri par bulle

Complexité

Soit *X, Y* et *Z* les dimensions de l'image 3D que l'on a à traiter et *n* la taille du filtre (taille du voisinage du pixel en cours de traitement, ou rayon). Bien évidemment, selon l'algorithme de tri de données que l'on utilise, la complexité change:

Pour le tri par bulle, la complexité du filtre médian est alors en

$$O\,(X\,.\,Y\,.\,Z.\,16\,.\,n^2\,(n^2 + 2n + 1))$$

Pour le tri Quicksort, la complexité du filtre médian est alors en

$$O\,(X\,.\,Y\,.\,Z\,.\,4n\,(n + 1)\,\log\,(4n\,(n + 1))\,)$$

5.1.1.3 Filtrage Gaussien

Applications

De la même manière que le filtre médian, les filtres gaussiens ont pour but de réduire le bruit présent dans une image, à ceci près qu'il a pour but de détecter les discontinuités de l'image. On l'utilise donc pour lisser une image, mais en tenant compte du fait que plus un pixel du voisinage est proche du pixel actuellement traité, plus il a de chances de faire partie du même objet, et donc d'avoir la même valeur que lui.

Principe

Chaque pixel d'un voisinage donné est doté d'un « poids », c'est-à-dire d'un coefficient pour le calcul de la moyenne. Plus le pixel est proche du pixel en cours de traitement, plus son poids est grand.
Pour réaliser un tel filtre, on utilise en fait une fonction que l'on appelle « gaussienne » (ou bien « fonction de Gauss », ou encore « Loi Normale » en statistiques). Cette fonction est de la forme suivante:

$$f\,(x\,,y\,,z) \;=\; \frac{1}{\left(\sqrt{2.\Pi}\,\sigma\right)^3} \;+\; \mathrm{e}^{-\frac{|x\,,y\,,z|^2}{2\sigma^2}}$$

On obtient donc le filtre gaussien et il ne nous reste plus qu'à l'appliquer à chaque pixel de l'image après avoir récupéré l'ensemble des points qui composent son voisinage.

Complexité

Soit *n* le nombre de voisins récupérés de chaque côté pour le calcul du filtre (c'est le rayon du filtre). Soit *X, Y* et *Z* les dimensions de l'image 3D traitée.

– La complexité pour le calcul du noyau du filtre gaussien est en

$$O((2n+1)^3)$$

– La complexité du calcul pour le filtrage de l'image 3D est en

$$O(X.Y.Z.\ 3\ (2n+1)^3)$$

La complexité totale de l'algorithme du filtre gaussien est donc en

$$O(3\ .\ X\ .\ Y\ .\ Z\ .\ (2n+1)^3 + (2n+1)^3\ ) \ = \ O(\ (2n+1)^3\ .\ (3.X.Y.Z+1)\ )$$

5.1.2 Extraction d'iso-surface

Applications

Les données en input ne correspondent pour l'instant qu'à une matrice de points présentent dans un tableau à trois dimensions. Après avoir appliqué les algorithmes de traitement d'images, on a modifié ces valeurs mais cela ne permet toujours pas une représentation visuelle en 3D. Ainsi, afin de pouvoir représenter un objet ou une zone qui serait intéressante à visualiser, on a besoin d'un algorithme spécifique permettant, à partir des valeurs en chaque voxel, de calculer la surface de l'objet ou de la zone en question. Un des algorithmes de calcul d'iso-surface effectuant très bien ce type de traitement se nomme le Marching Cubes. Nous allons maintenant en voir une description plus détaillée.

Principe

Pour fonctionner, l'algorithme du Marching Cubes a besoin de deux choses:

- Un tableau de points représentant un volume de données (de type températures, pression, densité d'absorption aux rayons X dans le cas d'un scanner, etc.), avec en chaque voxel une valeur associée.
- Une isovaleur, représentant la zone que l'on cherche à visualiser (du type 30 pour afficher une surface des températures égales à 30°C, ou bien différentes valeurs correspondantes à la densité d'absorption aux rayon X d'un certain organe du corps humain après après qu'un patient ait passé un scanner, etc.).

Si ces deux conditions sont réunies, il est alors possible d'afficher la zone correspondante à l'isovaleur passée en paramètre de l'application afin d'obtenir la représentation 3D adéquate. Cela fonctionne selon le principe suivant:

L'algorithme 'découpe' l'image en cubes de tailles égales et construit à l'intérieur de ces cubes des triangles qui, mis bout à bout, correspondront à la surface que l'on cherche à obtenir.

Techniquement, l'algorithme se sert de la valeur en chaque sommet du cube pour décider où tracer les triangles. Pour cela, il regarde pour tous les points du cube s'ils sont à l'intérieur de la surface (valeur d'un sommet inférieur à l'isovaleur passée en paramètre) ou à l'extérieur de la surface (valeur d'un sommet supérieure ou égale à l'isovaleur passée en paramètre), et trace le triangle en conséquence. Ainsi, comme un sommet a deux états possibles (intérieur / extérieur), et qu'il y a huit sommets dans un cube, cela donne $2^8 = 256$ possibilités pour tracer notre triangle.

Toute l'ingéniosité de cet algorithme repose dans la réduction de ces 256 possibilités à seulement 15 familles de possibilités. Cela vient du fait que certains cas sont les mêmes, à une rotation ou une symétrie près.

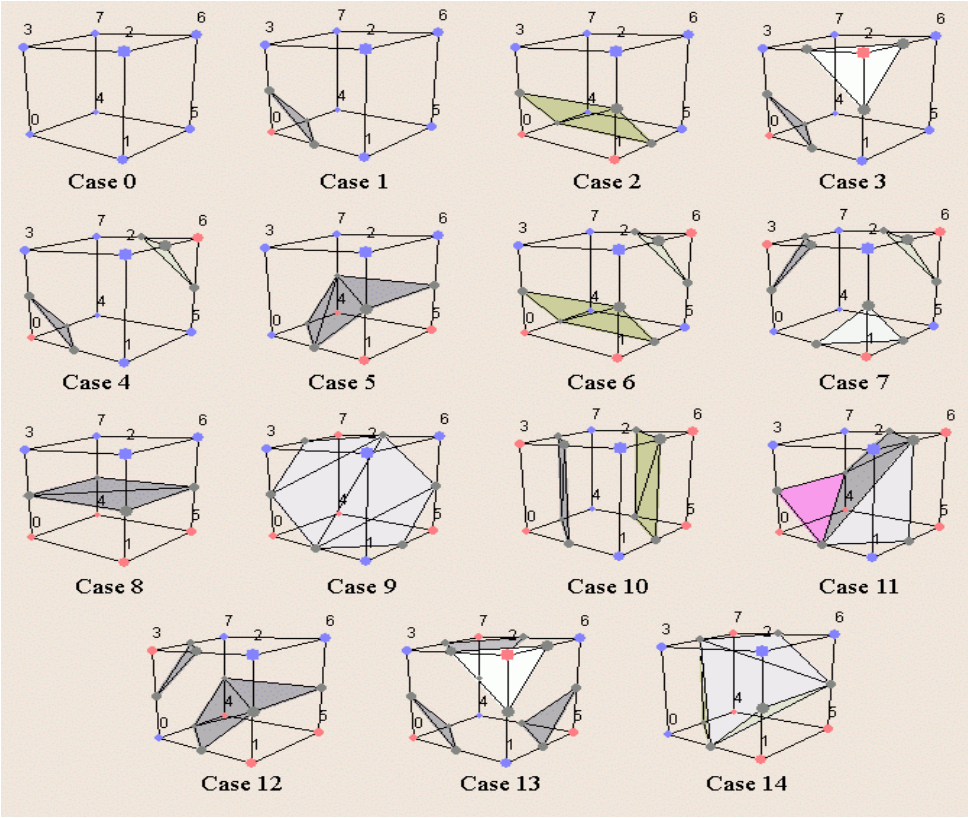


Figure 19. Les 15 familles de cas du Marching Cubes

L'algorithme du Marching Cubes peut donc se résumer à ces six étapes:

- Récupérer un cube dans l'image 3D avec ses valeurs associées à chaque sommet
- Regarder, pour chaque sommet du cube si il est à l'intérieur ou à l'extérieur de la surface que l'on chercher à représenter (i.e. en fonction de l'isovaleur passée en paramètre de l'algorithme)
- Regarder parmi les 15 possibilités à quel cas se réfère notre cube selon les sommets qui sont à l'intérieur ou à l'extérieur de la surface.
- Calculer les points des sommets du triangle où ceux-ci intersectent les arrêtes du cube. Pour cela, on effectue une interpolation linéaire de la manière suivante:

$$P = P_1 + \frac{(I - V_1)(P_2 - P_1)}{V_2 - V_1}$$

Où P_1 et P_2 sont les sommets du cube où un point du triangle intersecte l'arrête correspondante, V_1 et V_2 sont les valeurs associées à ces points et I est l'isovaleur.

- On passe au cube suivant
- Une fois tous les cubes traités, il ne reste plus qu'à assembler les résultats (triangles) obtenus afin d'obtenir l'iso-surface voulue

Complexité

Soit *n* le nombre d'arêtes utilisées pour former une arrête d'un cube. Soit *X, Y* et *Z* les dimensions de l'image 3D traitée. Soit V_S le nombre de sommets à l'intérieur de la surface que l'on cherche à représenter.

La complexité de l'algorithme du Marching Cubes est alors en

$$O(\frac{X \,.\, Y \,.\, Z}{n^3} + V_S)$$

5.1.3 Parallélisation

5.1.3.1 Description générale

Afin de réduire le temps de calcul, il est possible de modifier le code de l'application afin qu'il soit exécuté par plusieurs threads sur des serveurs multiprocesseurs à mémoire partagée. Ainsi, selon l'architecture du serveur que l'on utilise, il est possible de lancer plusieurs tâches qui, pour

êtres réellement efficaces ne doivent pas êtres plus nombreuses que le nombre de processeurs que contient la carte mère.
Par contre, il est possible de faire communiquer plusieurs serveurs entre eux afin de décupler le nombre de tâches. Ainsi, chaque serveur dont on se sert est capable de lancer un certain nombre de tâches (le nombre de processeurs contenus dans la carte mère), et on peut faire communiquer entre eux autant de serveurs que l'on veut. Cependant, si le nombre de serveurs qui communiquent est trop grand, la vitesse d'exécution d'une application risque de prendre plus de temps que si on avait utilisé moins de serveurs, ceci venant du fait que le nombre de messages échangés entre les différents serveurs augmente lui aussi en fonction du nombre de serveurs.

5.1.3.2 OpenMP

Comme nous venons de le voir, un programme séquentiel commun peut être modifier avec des directives OpenMP afin d'y introduire de la parallélisation multi-tâches sur des ordinateurs multiprocesseur. Le principe est simple: il suffit de définir quelles sont les variables *partagées* (communes et modifiables par tous les tâches, il faut alors faire attention aux conflits d'accès à la variable, souvent en y ajoutant un verrou pour qu'une seule tâche à la fois ne modifie la même variable), celles qui sont *privées* (qui sont privées pour chaque tâche et dont la valeur est initialisée au début des directives OpenMP) et celles qui sont '*firstprivate*' (qui sont privée pour chaque thread mais dont la valeur contenue est la même qu'avant la directive OpenMP), puis de déclarer le nombre de tâches que l'on souhaite lancer pour exécuter notre application. Bien sûr, il faut tenir compte du nombre de cœurs disponibles sur la machine sur laquelle on souhaite lancer notre application parallélisée, car sinon les tâches lancées en plus n'auront pas d'effet notable sur le temps de calcul puisque la parallélisation sera alors 'artificielle': un processus donne la main à un thread à la fois pendant que les autres sont bloqués.

5.1.3.3 MPI

Conçue en 1993, MPI est une norme définissant une bibliothèque de fonctions que l'on peut utiliser en Fortran ou, comme dans le cadre de ce stage, en C. Cette norme est devenue depuis le standard de communication pour des nœuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée. Le paradigme de programmation de MPI est le passage de message, fondamental pour comprendre son fonctionnement. Ainsi, dès qu'une machine (un processeur) désire communiquer avec un autre, il lui envoie un message.

MPI a été écrite pour obtenir de bonnes performances aussi bien sur des machines massivement parallèles à mémoire partagée que sur des clusters d'ordinateurs hétérogènes à mémoire distribuée. Elle est grandement disponible sur de très nombreux matériels et systèmes d'exploitation. Ainsi, MPI possède les avantages par rapport aux plus vieilles bibliothèques de passage de messages d'être grandement portables, car implantée sur presque toutes les architectures de mémoires, et rapide car chaque implantation a été optimisée pour le matériel sur lequel il

s'exécute. Tout cela vient du fait qu'elle est le produit, à la différence d'autres standards pour le développement d'applications parallèles, de réflexions entre des utilisateurs, des développeurs et des constructeurs. Ces derniers avaient en effet trop tendance à offrir avec leurs machines des outils de développement non portables. Comme MPI est un ensemble de spécifications et de fonctionnalités, elle ne fait aucune supposition sur le matériel sous-jacent. C'est cette indépendance qui lui a assuré son succès: de multiples implémentations sont disponibles, aussi bien libres que commerciales. Les premières visent la portabilité et le support des configurations hétérogènes tandis que les secondes ont pour objectif une exploitation optimale du matériel. Cependant, la majorité des implémentations de cette interface exhibent des défauts en ce qui concerne la réactivité face aux événements réseaux.

5.1.3.4 Application

Dans leurs locaux, les différents services d'IBM possèdent plusieurs serveurs multiprocesseurs qui sont dédiés à des activités diverses, comme la virtualisation, la gestion de bases de données, les services web, l'exécution d'applications 'memory-intensive' ou encore la recherche et le développement. Les différentes blades disponibles pour ces activités ont pour nom HS12, HS21, HS21 XM, LS20, LS21, LS22, LS41, LS42, JS12 Express, JS22 Express (pour les principales) et leurs performances et détails est explicité par la **figure 20** ci-dessous.

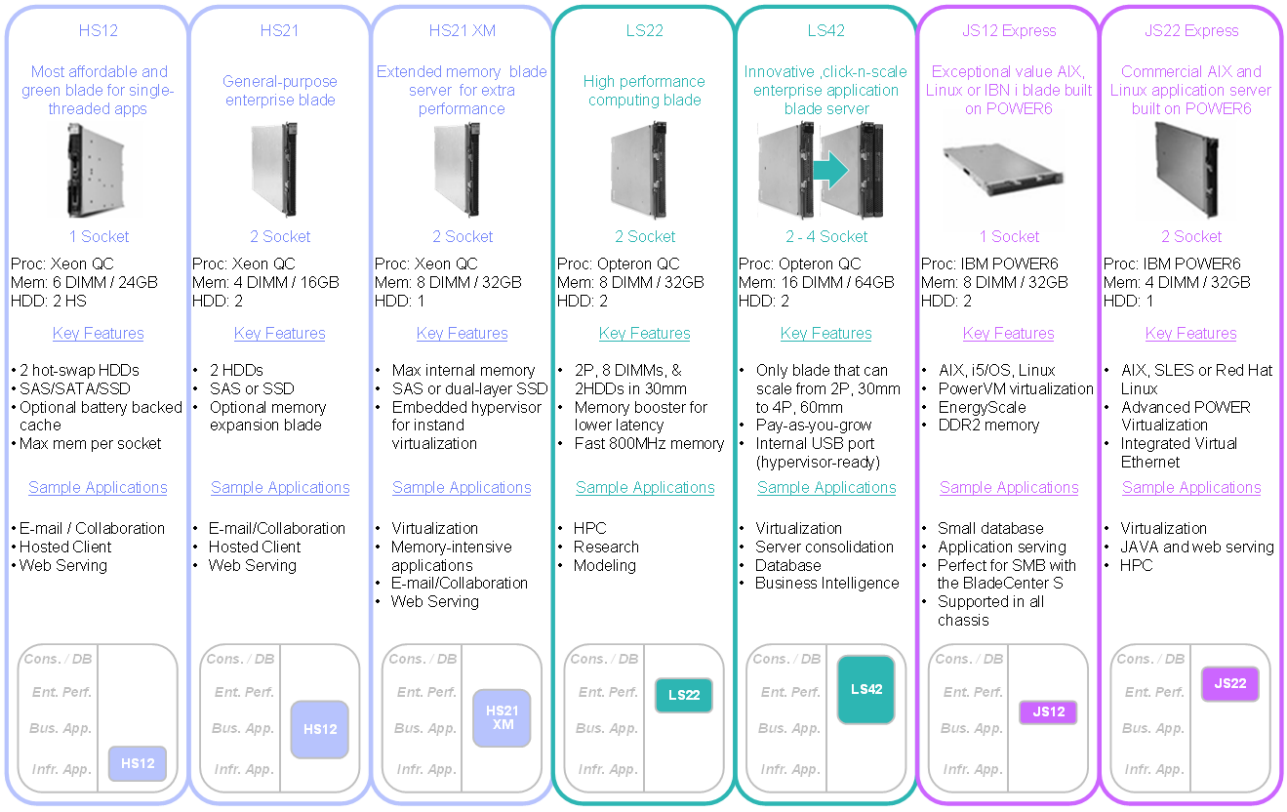


Figure 20. Comparaison des différentes blades d'IBM

Lors de ce stage, j'ai donc été amené à choisir un type de blade afin d'y faire tourner le code que j'avais écris pour qu'il puisse tourner en temps plus 'raisonnable' et d'y introduire de la parallélisation. Parmi les blades disponibles que nous avons testées, nous avons donc décidé de commencer par une blade LS22, disposant de 2 fois 4 cœurs, il était donc possible de lancer 8 threads (1 par processeur disponible). Le détail des performances d'une blade de type LS22, comparée aux performances des blades LS20, LS21, LS41, LS42 est explicité par la **figure 21** ci dessous.

	LS20	LS21	LS41	LS22	LS42
MTM	8850-xxx	7971-xxx	7972-xxx	7901-xxx	7902-xxx
Width	30mm	30mm	60mm	30mm	60mm
Chipset		HT2000/HT1000	HT2000/HT1000	HT2100/HT1000	HT2100/HT1000
Power Plane		Unified	Unified	Split	Split
Max # of Processors	2	2	4	2	4
Processor Models	AMD Opteron 2xx Single Core & Dual Core 68W	AMD Opteron 2xxx Dual Core & Quad Core 68W, 95W	AMD Opteron 8xxx Dual Core 68W, 95W	AMD Opteron 23xx Quad Core 79W, 115W	AMD Opteron 83xx Quad Core 79W, 115W
ax Proc SKU	2.4GHz 68W Dual Core	3.0GHz Dual Core 2.6GHz 68W Dual Core	3.0GHz 95W Dual Core 2.6GHz 68W Dual Core	2.3GHz 115W Quad Core 1.9GHz 79W Quad Core	2.3GHz 115W Quad Cor 1.9GHz 79W Quad Core
Max Memory	4 DIMMs / 8GB DDR1 up to 400MHz	8 DIMMs / 32GB VLP DDR2 up to 667MHz	16 DIMMs / 64GB VLP DDR2 up to 667MHz	8 DIMMs / 64GB VLP DDR2 up to 800MHz	16 DIMMs / 128GB VLP DDR2 up to 800MH
Storage	2x fixed SCSI	1x fixed SFF SAS	2x fixed SFF SAS	2x fixed SFF SAS or 2x SSD	2x fixed SFF SAS or 2x SSD
Imbed USB	No	No	No	YES	YES
Ethernet	2x Gigabit	2x Gigabit w/TOE	4x Gigabit w/TOE	2x Gigabit w/TOE IPv6	4x Gigabit w/TOE IPv6
I/O	1 PCI-X	1 PCI-X and 1 PCI-E	2 PCI-X and 1 PCI-E	1 PCI-X and 1 PCI-E	2 PCI-X and 1 PCI-E
Management	BMC w/IPMI 1.5	BMC w/IPMI 2.0	BMC w/IPMI 2.0	BMC w/IPMI 2.0	BMC w/IPMI 2.0
Warranty	3 yr	3 yr	3yr	3yr	3yr

Figure 21. Performances des blades LS20, LS21, LS22, LS41 et LS42

Pour simplifier, on peut dire que le blade LS22 a les caractéristiques suivantes:

- *CPU:* Carte mère avec 2 sockets de 4 processeurs à 2,7 GHz.
- *Mémoire:* 2 fois 4 barrettes de mémoire RAM à 2 GB.

Et voici comment on pourrait représenter schématiquement son architecture:

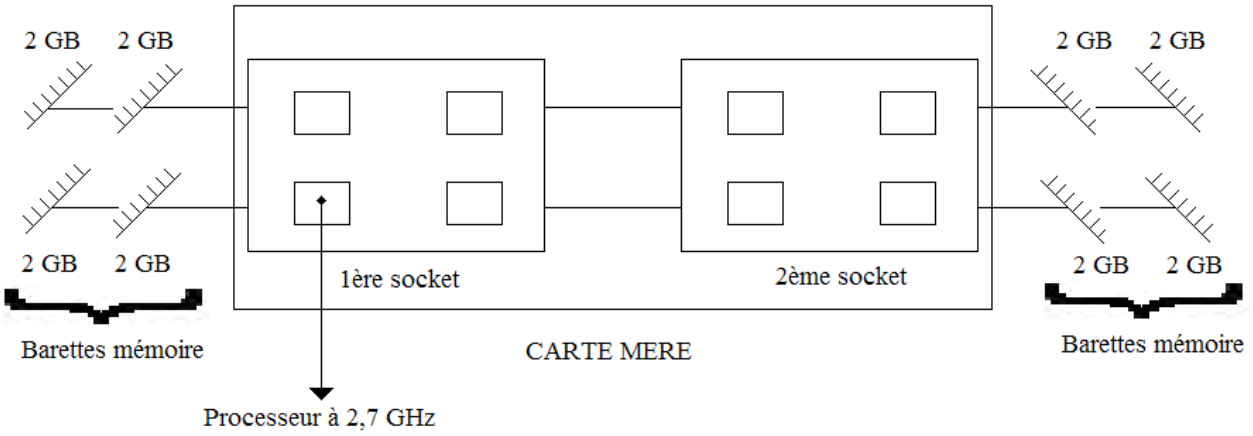


Figure 22. Architecture simplifiée d'une blade LS22

La parallélisation étant implémentée sur cette blade, cela a permis de réduire significativement le temps de calcul, et les résultats obtenus sont tous présentés dans les sections **4.1** Résultats obtenus et **5.1.5** Benchmark.

5.1.4 OpenGL

Le système graphique OpenGL, de Silicon Graphics Inc. (SGI), s'est imposé comme la bibliothèque de référence pour réaliser des projets de synthèse d'images en temps réel dans le milieu professionnel. Face à son grand succès sur les stations Silicon Graphics, plateforme sur laquelle elle peut être considérée comme succédant à la bibliothèque IRISGL, cette librairie s'est maintenant généralisée sur de nombreux systèmes d'exploitation. Des clones de cette fameuse librairie sont d'ailleurs disponibles sur certains systèmes d'exploitation avant que l'originale ne soit portée. Le plus connu de ces clones est Mesa de BRIAN Paul, disponible sous Linux, Windows et plus récemment sur Macintosh. S'appuyant sur leur expérience des bibliothèques de fonctions 3D, SGI a voulu proposer un produit facile à comprendre et à utiliser. Par rapport à IRISGL, leur ancienne API, de nettes améliorations ont été apportées :

- Elle est multi plateforme par sa conception.

- Inspirée par l'interface graphique XWindow, elle propose un modèle client-serveur sur réseau hétérogène où le serveur effectue les calculs et le client l'affichage. Ainsi, le serveur peut être un supercalculateur qui exécute des calculs complexes de simulation alors que le client est un poste de travail principalement utilisé pour ses capacités de visualisation. Cependant, il est bien sûr possible que tout soit traité sur la même machine.

En plus d'être multi plateforme, la librairie OpenGL peut être utilisée depuis d'autres langages de programmation que le langage C : Fortran, Ada, C++ et Java. Cette caractéristique fait d'OpenGL la librairie graphique utilisable sur le plus grand nombre d'environnements différents. Comparée à Quickdraw3D et Direct3D, OpenGL n'est pas une API de haut niveau dans le sens où elle ne fournit pas de fonctions pour effectuer des tâches de haut niveau comme l'édition d'objet ou la gestion de fichiers. Ces tâches sont laissées à la charge du programmeur. Pour manipuler une API de haut niveau exploitant OpenGL, le meilleur choix est d'écrire une application utilisant Silicon Graphics Open Inventor. Cette API propose en effet une architecture 3D orientée objets donnant accès à des fonctions de haut niveau. Cette API, de ce fait, est fortement conseillée pour les novices en programmation 3D et ceux qui préfèrent la structuration des programmes orientés objets. Un point très important dans le développement de cette bibliothèque est que les cartes graphiques avec fonctions 3D câblées prennent désormais en charge au niveau matériel les quelques 250 commandes OpenGL. Jusqu'à récemment de telles cartes coûtaient très cher et étaient uniquement disponibles sur certaines stations de travail UNIX dont les Silicon Graphics. Actuellement, grâce à la volonté d'ouverture de SGI, toutes les cartes d'entrée de gamme supportent désormais en plus des commandes Direct3D, celles d'OpenGL. L'effort des constructeurs va même jusqu'à proposer des drivers pour Linux, preuve qu'ils croient à la fois au développement d'OpenGL et à celui de Linux.

Pour les cartes à base de processeurs graphiques Voodoo de 3Dfx Interactive, première famille de carte à avoir proposé un driver sous Linux, cette accélération se fait par l'intermédiaire de la librairie Glide qui sert d'interface vers les registres de la carte.

La **Figure 23** nous montre d'une part l'architecture logicielle d'OpenGL et, d'autre part, l'ordre des opérations dans son pipeline graphique. Puisque l'architecture d'OpenGL est de type client-serveur, les calculs et l'affichage de la scène pourront soit être faits sur une même machine, soit répartis selon les possibilités des machines. Dans ce second cas, le pipeline graphique sera géré par le serveur qui recevra les données du client et lui renverra la scène calculée. C'est dans le pipeline graphique que le système détermine si une commande est accélérée par le matériel. Selon le cas, l'exécution de cette commande sera aiguillée vers un module matériel ou un module d'émulation logicielle.

OpenGL compte deux modes de rendu différents : un mode immédiat où les commandes sont envoyées une à une au serveur que celui-ci exécute le plus rapidement possible et un mode retenu où les commandes sont stockées dans une liste et envoyées en bloc au serveur. Ce dernier mode compte deux avantages importants : pour afficher un objet complexe, il suffit de référencer la liste au lieu de redonner toutes les commandes, les informations sur un objet sont ainsi envoyées rapidement sur le réseau.

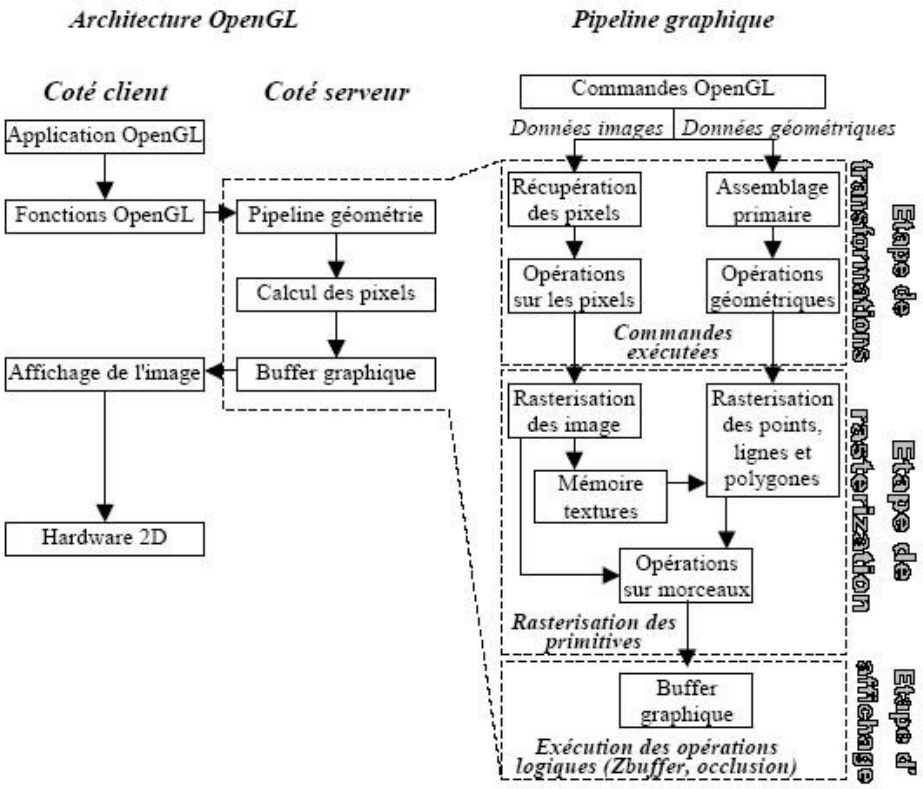


Figure 23. Architecture logicielle et pipeline graphique d'OpenGL.

Le seul inconvénient de ce mode retenu est lorsque l'on souhaite modifier l'objet. Il faudra pour cela supprimer la liste actuelle et en créer une autre contenant l'objet modifié. Avec OpenGL, il est cependant possible de mélanger des commandes immédiates et des exécutions de liste de commandes, rendant ainsi possible la gestion des objets animés en mode immédiat et ceux fixes en mode retenu.

OpenGL semble, pour toutes ces qualités, une API de choix pour la création d'une application 3D. Les seuls points négatifs par rapport à ses concurrents est le manque d'architecture orientée objets et le fait que les textures utilisées doivent avoir des dimensions de 2n pixels en largeur et de 2m pixels en hauteur pour les raisons évidentes d'un rendu plus rapide car les images peuvent ainsi être parcourues pixel à pixel par décalages au lieu de multiplications plus coûteuses.

5.1.5 Benchmark

Intégration des données

Taille de l'image en x, y et z	Nombre de voxels	Temps d'exécution en séquentiel (en secondes)	Temps d'exécution avec directives OpenMP (en secondes)	Rapport séquentiel / parallélisé
100 *x* 100 *x* 100	1.000.000	0,12	0,1	1,2
500 *x* 500 *x* 100	25.000.000	0,49	0,31	1,6
500 *x* 500 *x* 300	75.000.000	1,38	0,67	2,1
500 *x* 500 *x* 500	125.000.000	2,22	1,11	2
1000 *x* 1000 *x* 500	500.000.000	9,15	4,38	2,1
1000 *x* 1000 *x* 1000	1.000.000.000	18,2	7,9	2,3
2000 *x* 1000 *x* 1000	2.000.000.000	36,6	15,2	2,4
2780 *x* 836 *x* 907	2.110.264.640	303 = 5mn 3s	78 = 1mn 18s	3,9
2000 *x* 1000 *x* 1375	2.750.000.000	51,1	20,7	2,5

Table 1. Temps d'exécution pour intégrer les données

Mesures effectuées sur une blade de type LS22

Filtre Médian

Taille de l'image en x, y et z	Nombre de voxels	Temps d'exécution en séquentiel (en secondes)	Temps d'exécution avec directives OpenMP (en secondes)	Rapport séquentiel / parallélisé
100 *x* 100 *x* 100	1.000.000	2,56	0,32	8
500 *x* 500 *x* 100	25.000.000	67,65 = 1mn 7s	9,8	6,9
500 *x* 500 *x* 300	75.000.000	193,77 = 3mn 13s	28,4	6,8
500 *x* 500 *x* 500	125.000.000	287,3 = 4mn 47s	46,4	6,2
1000 *x* 1000 *x* 500	500.000.000	1263,6 = 21mn 4s	193,4 = 3mn 13s	6,5
1000 *x* 1000 *x* 1000	1.000.000.000	2507 = 41mn 47s	361 = 6mn 11s	6,9
2000 *x* 1000 *x* 1000	2.000.000.000	5195 = 86mn 35s	769 = 12mn 49s	6,8
2780 *x* 836 *x* 907	2.110.264.640	5317 = 88mn 37s	892 = 14mn 52s	6
2000 *x* 1000 *x* 1375	2.750.000.000	7221 = 120mn 1s	1107 = 18mn 27s	6,5

Table 2. Temps d'exécutions du filtre médian

Mesures effectuées sur une blade de type LS22

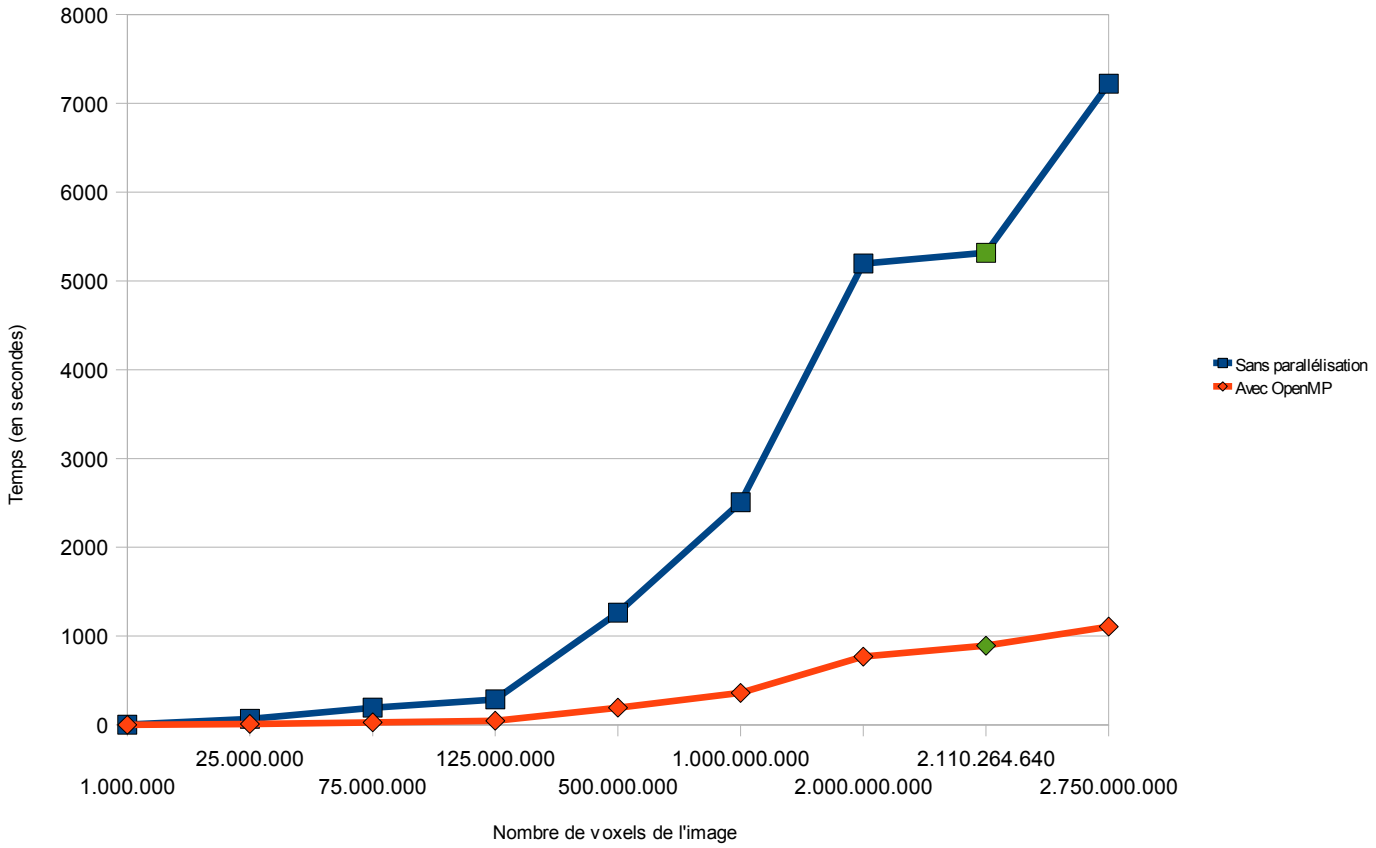


Figure 24. Temps d'exécution du filtre médian

Filtre Gaussien

Taille de l'image en x, y et z	Nombre de voxels	Temps d'exécution en séquentiel (en secondes)	Temps d'exécution avec directives OpenMP (en secondes)	Rapport séquentiel / parallélisé
100 *x* 100 *x* 100	1.000.000	0,72	0,13	5,5
500 *x* 500 *x* 100	25.000.000	16,62	1,6	10,4
500 *x* 500 *x* 300	75.000.000	50,3	5,6	9
500 *x* 500 *x* 500	125.000.000	78,1	9,7	8
1000 *x* 1000 *x* 500	500.000.000	342 = 5mn 42s	41,7	8,2
1000 *x* 1000 *x* 1000	1.000.000.000	669 = 11mn 9s	72 = 1mn 12s	9,3
2000 *x* 1000 *x* 1000	2.000.000.000	1341 = 22mn 21s	155 = 2mn 35s	8,7
2780 *x* 836 *x* 907	2.110.264.640	1408 = 23mn 28s	170 = 2mn 50s	8,3
2000 *x* 1000 *x* 1375	2.750.000.000	1833 = 30mn 33s	230 = 3mn 50s	8

Table 3. Temps d'exécution du filtre gaussien

Mesures effectuées sur une blade de type LS22

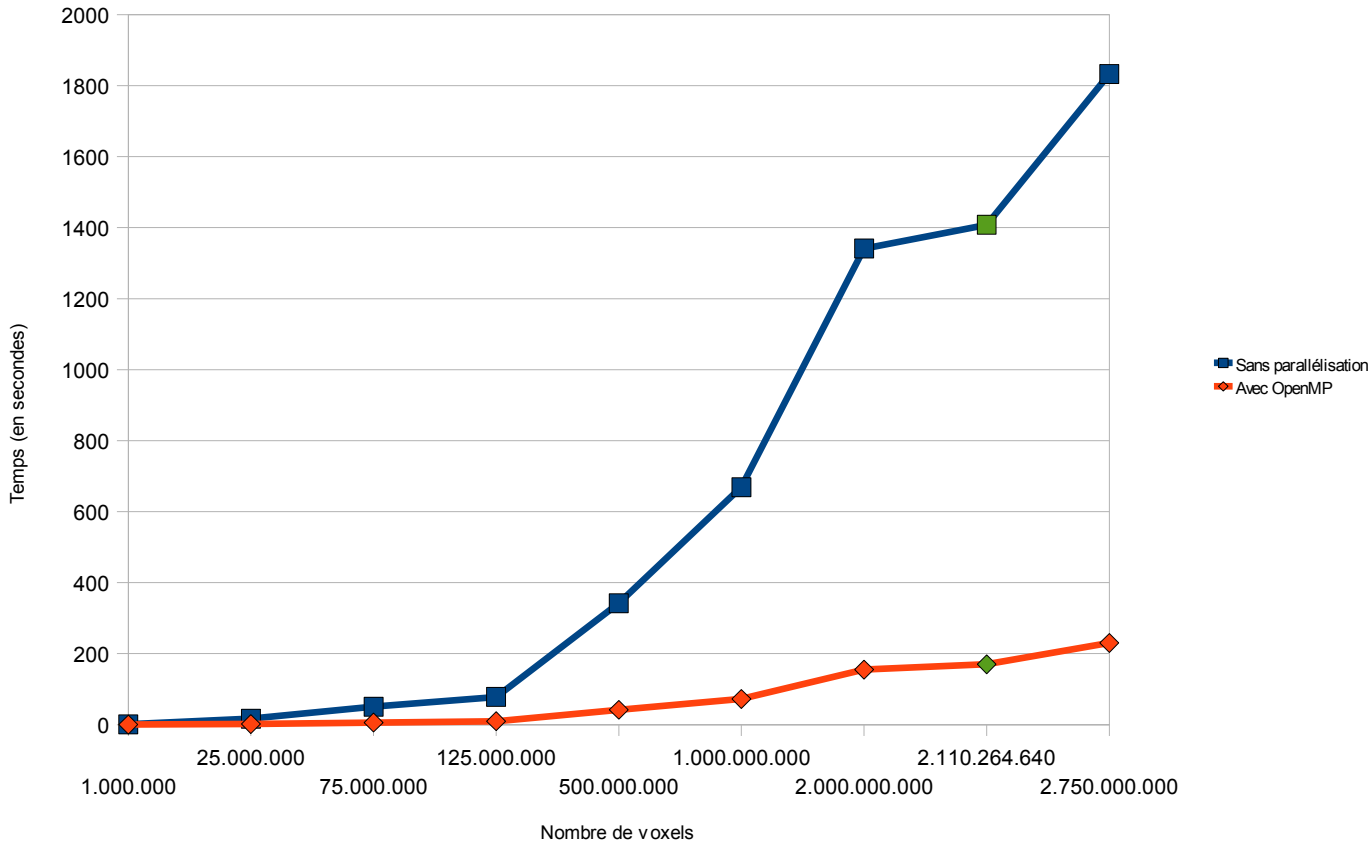


Figure 25. Temps d'exécution du filtre gaussien

Marching cubes

Taille de l'image en x, y et z	Nombre de voxels	Temps d'exécution en séquentiel (en secondes)	Temps d'exécution avec directives OpenMP (en secondes)	Rapport séquentiel / parallélisé	Nombre de triangles
100 *x* 100 *x* 100	1.000.000	0,23	0,12	1,9	14.690
500 *x* 500 *x* 100	25.000.000	4,9	0,88	5,6	406.981
500 *x* 500 *x* 300	75.000.000	14,1	2,3	6,1	1.615.667
500 *x* 500 *x* 500	125.000.000	26,4	4	6,6	3.299.681
1000 *x* 1000 *x* 500	500.000.000	102,2 = 1mn 42s	18,7	5,5	20.873.338
1000 *x* 1000 *x* 1000	1.000.000.000	246,9 = 4mn 7s	49,3	5	53.223.168
2000 *x* 1000 *x* 1000	2.000.000.000	570 = 9mn 30s	107,9 = 1mn 48s	5,3	137.013.951
2780 *x* 836 *x* 907	2.110.264.640	614 = 10mn 14s	90 = 1mn 30s	6,8	37.943.758
2000 *x* 1000 *x* 1375	2.750.000.000	785 = 13mn 5s	167,5 = 2mn 8s	4,7	212.022.609

Table 4. Temps d'exécution du Marching cubes

Mesures effectuées sur une blade de type LS22

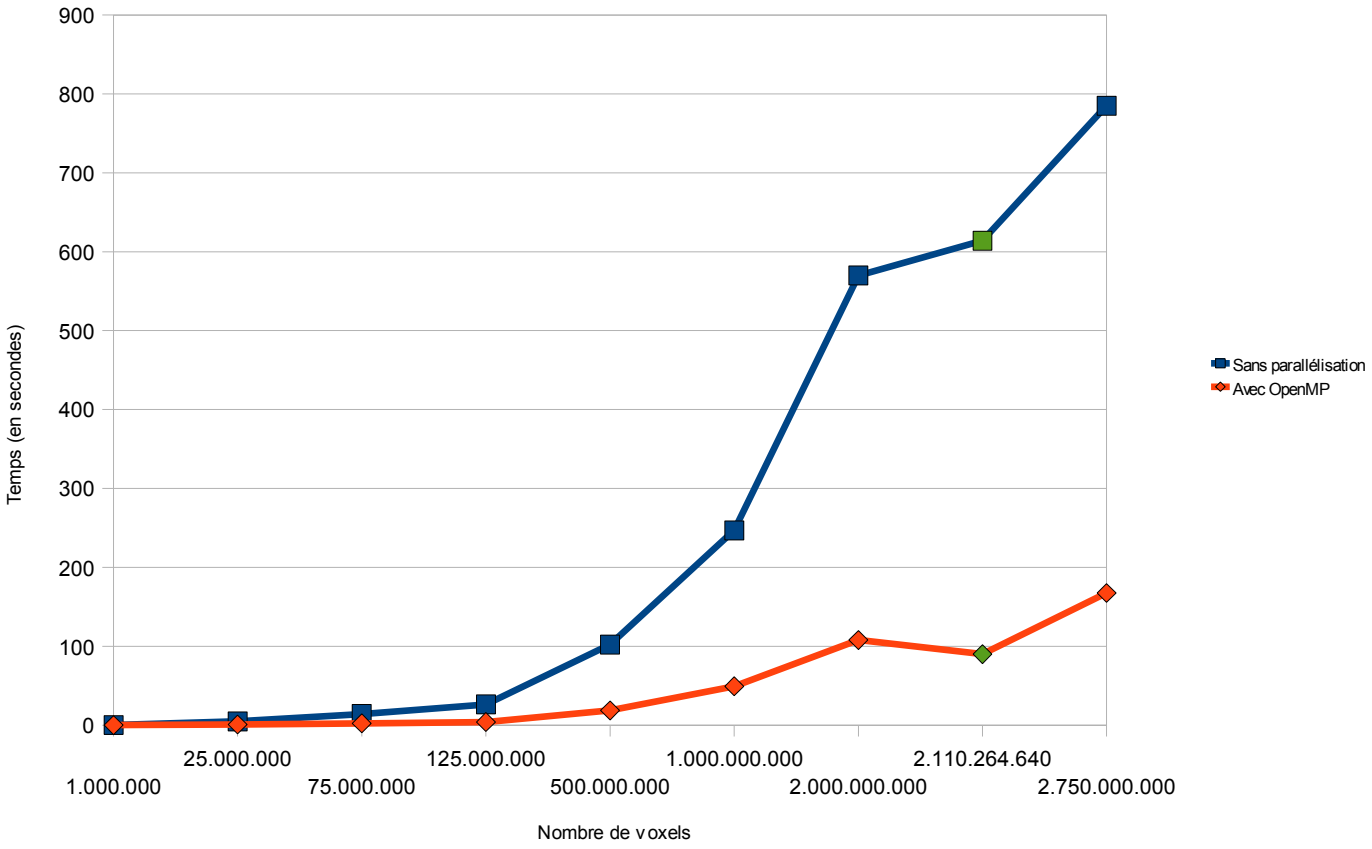


Figure 26. Temps d'exécution du Marching cubes

5.2 Table des figures

5.3 Glossaire

API: Outil logiciel, constitué d'un ensemble de modules dotés de fonctions communes, qui permet d'ajouter un ensemble de fonctionnalités données pour un certain langage de programmation.

Cluster: Grappe de machines inter-connectées et vues comme un même ensemble de traitement, apportant des fonctions de disponibilité, de répartition de charge et de partage des données

Gprof: c'est un logiciel GNU Binary Utilities qui permet d'effectuer du profilage de code.

GTK: The GIMP Toolkit, c'est un ensemble de bibliothèques logicielles, c'est-à-dire un ensemble de fonctions informatiques, permettant de réaliser des interfaces graphiques. Cette bibliothèque a été développée originellement pour les besoins du logiciel de traitement d'images GIMP. GTK+ est maintenant utilisé dans de nombreux projets, dont les environnements de bureau GNOME, Xfce et ROX.

Iso-surface: surface qui représente des points d'une valeur constante à l'intérieur d'un volume de l'espace. En d'autres termes, c'est un niveau fixé d'une fonction continue dont le domaine est dans l'espace 3D.

MPI: The Message Passing Interface. Conçue en 1993, c'est une norme définissant une bibliothèque de fonctions, utilisables avec les langages C et Fortran, qui permet d'exploiter des ordinateurs distants ou multiprocesseur par passage de messages.

OpenMP: Open Multi Processing**,** c'est une API pour les plateformes multiprocesseurs à mémoire partagée afin de paralléliser des programmes écris en C/C++ et Fortran.

OpenGL: Open Graphics Library, c'est une spécification qui définit une API multi-plateforme pour la conception d'applications générant des images 3D (mais également 2D). Elle utilise en interne les représentations de la géométrie projective pour éviter toute situation faisant intervenir des infinis.

OS: Operating System, c'est un ensemble de programmes responsables de la liaison entre les ressources matérielles d'un ordinateur et les applications informatiques de l'utilisateur (traitement de texte, jeux vidéo, etc…). Il fournit aux programmes d'applications des points d'entrée génériques pour les périphériques.

ThinkPad: ordinateur personnel standard fournit par IBM, du même nom que la société qui les fabriquent.

5.4 Bibliographie

Algorithmes de traitement d'images:

[1] http://www.ensta.fr/~manzaner/Support_Cours.html

[2] http://xphilipp.developpez.com/articles/filtres/

[3] http://stfsworld.com/article20.html

[4] http://www.siteduzero.com/tutoriel-3-80697-les-filtres-de-convolution.html

MicroView:

[5] http://www.gehealthcare.com/usen/fun_img/pcimaging/products/microview.html

Makefile:

[6] http://ftp.traduc.org/doc-vf/gazette-linux/html/2002/083/lg83-B.html

Marching Cubes:

[7] http://www.ia.hiof.no/~borres/cgraph/explain/marching/p-march.html

[8] http://www.cppfrance.com/codes/MARCHING-CUBE-MARCHING-TETRAHEDRA_17865.aspx

[9] http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html

[10] http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/

Parallélisation:

[11] http://www.idris.fr/data/cours/parallel/openmp/IDRIS_OpenMP_cours.pdf

[12] https://computing.llnl.gov/tutorials/openMP/

[13] http://www.u-cergy.fr/sir/data/coursOpenMP/coursOpenMP-HTML/index.html