

N° d'ordre : 1701
EDSPIC : 361

UNIVERSITÉ BLAISE PASCAL - CLERMONT II

*École Doctorale
Sciences Pour l'Ingénieur de Clermont-Ferrand*

Thèse présentée par :
Joël FALCOU

Formation Doctorale CSTI :
Composants et Systèmes pour le Traitement de l'Information
en vue de l'obtention du grade de

DOCTEUR D'UNIVERSITÉ

spécialité: Vision pour la Robotique

Un *cluster* pour la Vision Temps Réel Architecture, Outils et Applications

Soutenue publiquement le : 1^{er} Décembre 2006 devant le jury :

Monsieur Jean-Thierry LAPRESTÉ	Président du Jury
Monsieur Jocelyn SÉROT	Directeur de thèse
Monsieur Daniel ETIEMBLE	Rapporteur
Monsieur Frédéric LOULERGUE	Rapporteur
Monsieur Thierry CHATEAU	Examinateur
Monsieur Franck CAPPELLO	Examinateur

Remerciements

Je tiens tout d'abord à remercier Jocelyn Sérot pour la confiance qu'il m'a accordé quant au déroulement de ces travaux. Sa disponibilité et sa gentillesse m'ont permis de mener de manière relativement libre cette thèse. Il m'a aussi soutenu dans mes périodes de doutes et su m'insuffler la motivation nécessaire pour aller de l'avant.

Je remercie également Thierry Chateau qui a su m'apporter les connaissances nécessaires en vision pour me permettre de mener mes travaux à bien. Un grand merci à Jean Thierry Lapresté pour m'avoir supporté – peut être dans tous les sens du terme – au cours de ces trois années.

Je remercie aussi Messieurs les professeurs Daniel Etiemble et Frédéric Loulergue pour m'avoir fait l'honneur de relire ce manuscrit et d'avoir participer au Jury. Ils ont également contribué par leurs remarques et suggestions à améliorer la qualité de ce mémoire, et je leur en suis très reconnaissant.

Messieurs Franck Cappello et Frédéric Jurie m'ont fait l'honneur de participer au Jury de soutenance et je les en remercie profondément.

Je me dois aussi de remercier Alice, Alexandre, Christophe, Loïc, Brice, Eric, Lucie, Etienne, Steve, Lionel, François, Laetitia, Fabio, Sébastien, Hicham, Pierre, François, Christophe pour leur amitié et leur aide dans les moments difficiles.

Enfin, pour leurs encouragements et leur assistance aussi bien matérielle que morale qui m'ont permis d'effectuer cette thèse dans de bonnes conditions, je tiens à remercier mes parents et ma femme qui ont porté sur leur épaules la lourde tâche de me supporter.

Résumé

Les besoins en terme de puissance de calcul des applications de vision artificielle n'ont cessé d'augmenter avec le développement de nouvelles théories et méthodes, alors que leur temps d'exécution se devait de rester compatible avec la cadence des capteurs images (*i.e.* 25 à 30 images/s).

Les travaux présentés dans ce manuscrit se proposent d'apporter une solution logicielle permettant de mettre au point de telles applications et de les exécuter à une fréquence compatible avec le temps-réel vidéo sur des machines de type *cluster*. Pour cela, nous avons développé des bibliothèques haut-niveau permettant de programmer ces machines de manière efficace, en s'attachant à rendre accessible les modèles de programmations parallèles à des développeurs issus de la communauté Vision pour qui ces problématiques ne sont pas triviales. La difficulté est ici de conserver un niveau de performance élevé malgré une interface de haut-niveau. Pour cela, des techniques avancées de génie logiciel comme l'évaluation partielle et la métaprogrammation *template* ont permis de développer deux bibliothèques : E.V.E. , qui propose une interface proche de celle de MATLAB® et qui prend en charge la gestion du parallélisme SIMD, et QUAFF qui se base sur un modèle de programmation à base de squelettes algorithmiques afin de faciliter le développement sur des machines MIMD.

Ces travaux sont validés par deux applications de vision de complexité réaliste - une reconstruction 3D temps réel et un suivi de piéton par filtrage particulaire - développées et exécutées sur un *cluster* – la machine BABYLON – équipée de deux bus de communications et de quatorze noeuds de calcul biprocesseurs exhibant trois niveaux de parallélisme : MIMD, SMP et SIMD. Sur cette architecture, des accélérations de l'ordre de 30 à 100 par rapport à une implantation séquentielle classique ont été mesurées.

Mots-clés :Parallélisme, Vision artificielle temps-réel, Cluster, Évaluation partielle, Méta-programmation template, Squelettes algorithmiques.

Abstract

Recent works in computer vision produced complex algorithms and applications that require more and more computing power despite the fact that they need to be run at real-time frequency (usually around 25 to 30 images per second).

This thesis presents a novel software solution for developing such applications and to run them in real-time on *cluster* machines. To do so, we developed efficient, high-level object-oriented libraries while making the usual parallel programming model easier to use for computer vision developers. The main difficulty is to find a good trade-off between efficiency and readability. To do so, we used advanced software engineering techniques to implement two object-oriented libraries : E.V.E. , that provides a MATLAB® like interface to take care of SIMD parallelism, and QUAFF , which is an object-oriented implementation of the parallel algorithmic skeletons model.

This work has been validated with two realistic computer vision applications – a real time 3D reconstruction and a particule filter based pedestrian tracker – that were developed and run on a *cluster* – the BABYLON machine – using two communication bus and fourteen computing nodes with two processors, exhibiting three level of parallelism : MIMD, SMP and SIMD. On this architecture, speed-up of 30 to 100 – compared to basic implementation – have been measured.

Keywords: Parallelism, Real-time computer vision, Cluster, Partial evaluation, Template metaprogramming, Algorithmique skeletons.

Table des matières

Introduction	1
1 Architectures dédiées à la vision	9
1.1 Les Clusters	9
1.1.1 NOW : Networks Of Workstations	11
1.1.2 Beowulf	11
1.1.3 Blue Gene	12
1.2 Les Clusters pour la vision	13
1.2.1 <i>Virtualized Reality - 3D ROOM</i>	14
1.2.2 GrImage	15
1.2.3 ViROOM	16
1.2.4 Beobots	17
1.2.5 OSSIAN	18
1.3 Réseaux logiques programmables	19
1.4 Cartes d'accélération graphique	19
1.5 Conclusion	20
2 La machine BABYLON	23
2.1 Architecture matérielle	24
2.1.1 Noeud de calcul	24
2.1.2 Réseau de communication	24
2.1.3 Topologie des réseaux de communications	26
2.2 Synopsis général	29
2.2.1 Architecture du POWER PC G5	31
2.2.2 Spécification du processeur PPC 970	33
2.2.3 Spécifications de l'extension SIMD ALTIVEC	35
2.2.4 Conclusion	37
2.3 La bibliothèque d'acquisition vidéo C+FOX	38
2.3.1 Interface utilisateur	38
2.3.2 Gestion de la multi-diffusion	39

2.4	Outil de développement MIMD	41
2.4.1	Principes de MPI	41
2.4.2	Exemple d'utilisation	43
2.4.3	Conclusion	44
2.5	Outils de développement pour architectures SMP	44
2.5.1	MPI	44
2.5.2	Open MP	45
2.5.3	pThread	46
2.5.4	Conclusion	49
2.6	Outil de développement SIMD	50
2.6.1	Performances	52
2.6.2	Mise en oeuvre	53
2.6.3	Conclusion	57
2.7	Application : Stabilisation de flux vidéo	58
2.7.1	Description de l'algorithme	59
2.7.2	Implantation séquentielle	60
2.7.3	Stratégie de parallélisation	65
2.7.4	Performances	67
2.7.5	Conclusion	70
2.8	Modèle de performance	71
2.8.1	Loi d'Amdahl	71
2.8.2	Loi de Gustafson-Barsis	72
2.8.3	Modèle de performance hybride	73
2.8.4	Mise en oeuvre	74
2.9	Conclusion	75
3	Outils logiciels pour le calcul parallèle	77
3.1	Modèles et outils pour la programmation SIMD	79
3.1.1	<i>L'Apple Accelerate.Framework</i>	79
3.1.2	VAST	80
3.2	Modèles et outils pour la programmation MIMD	80
3.2.1	Les squelettes algorithmiques	81
3.2.2	<i>Les Design Patterns</i>	85
3.2.3	Autres approches	87
3.3	Discussion	88
3.3.1	Approches basées sur des nouveaux langages	88
3.3.2	Approches basées sur des générateurs de code	89
3.3.3	Approches basées sur des bibliothèques	89
3.3.4	Solution proposée	90

4 La bibliothèque E.V.E.	91
4.1 Modèle de programmation	91
4.1.1 Problématique de l'implantation orientée objet	94
4.2 Interface utilisateur	96
4.2.1 Les classes <code>array</code> et <code>view</code>	96
4.2.2 Choix des options d'optimisations	97
4.2.3 Fonctions disponibles	98
4.3 Implantation	102
4.3.1 L'évaluation partielle en C++	102
4.3.2 Les <i>Expression Templates</i>	105
4.3.3 Détermination de la «vectorisabilité» d'une expression . .	115
4.3.4 Optimisations spécifiques	120
4.4 Évaluations des performances	121
4.4.1 Accélération SIMD	126
4.4.2 Optimisation des compositions d'opérateurs	127
4.5 Étude de cas	128
4.5.1 Présentation de l'algorithme	128
4.5.2 Implantations séquentielles	130
4.5.3 Implantations optimisées	132
4.5.4 Discussion	134
4.6 Conclusion	135
5 La bibliothèque QUAFF	137
5.1 Modèle de programmation	138
5.1.1 Définition formelle d'une application	141
5.1.2 Squelettes dédiés au parallélisme de contrôle	142
5.1.3 Squelettes dédiés au parallélisme de données	143
5.1.4 Squelettes dédiés à la structuration de l'application . .	146
5.1.5 Exemple de mise en œuvre	147
5.2 Interface utilisateur	148
5.2.1 Définition des tâches séquentielles	148
5.2.2 Interfaces des squelettes	149
5.2.3 Définition des entrées/sorties	151
5.2.4 Définition d'une application QUAFF	153
5.3 Implantation	154
5.3.1 Manipulation des listes de fonctions	155
5.3.2 Mise en œuvre du parallélisme	161
5.3.3 Gestion des communications	174
5.4 Validation	177

5.5	Conclusion	179
6	Applications à la stéréovision	181
6.1	Principe de la stéréo-vision	182
6.1.1	Modèle sténopé des caméras	183
6.1.2	Principes géométrique de la stéréo-vision	184
6.2	Reconstruction 3D temps réel	185
6.2.1	Implantation séquentielle	186
6.2.2	Implantation parallèle	197
6.2.3	Résultats	202
6.3	Détection et suivi 3D temps réel de piétons	205
6.3.1	Approche probabiliste du suivi d'objets	205
6.3.2	Le filtre à particules	206
6.3.3	Implantation séquentielle	211
6.3.4	Implantation parallèle	212
6.3.5	Résultats	218
6.4	Conclusion	219
Conclusion et perspectives		221
Annexes		225
I	<i>templates</i> et métaprogrammation	227
I.1	Définitions	227
I.1.1	Classe <i>template</i>	227
I.1.2	Fonction <i>template</i>	228
I.1.3	Spécialisation des <i>templates</i>	229
I.2	Principes de métaprogrammation	230
I.2.1	Arithmétique statique	230
I.2.2	Structure de contrôle statique	231
I.2.3	Fonctions de manipulations de types	232
I.3	Méta-programmes usuels	234
I.3.1	Adaptateur valeur/type	234
I.3.2	Opérateurs logiques statiques	234
I.3.3	Test de conversion implicite	236
I.3.4	Levée d'erreurs à la compilation	239
I.4	Discussion	240

II prime_sieve par E. Unruh	241
III Exemple de fonction ALTIVÉC complexe	243
IV Opérations sur les listes statiques de types	245
IV.1 Evaluation de la taille d'une liste de type	246
IV.2 Accès aléatoire aux éléments d'une liste de type	246
IV.3 Application d'une métा-fonction à une liste	247
IV.4 Retrait d'élément à une liste de type	247
IV.5 Ajout d'élément à une liste de type	248
IV.6 Extraction d'une sous-liste	249
IV.7 Accès à l'élément d'un tuple	250
Bibliographie	250

Table des figures

1	Classification de Flynn-Johnson	4
1.1	Architecture générique d'un <i>cluster</i> extraite de [33]	10
1.2	Deux segments du <i>cluster</i> NOW	11
1.3	Architecture du <i>cluster</i> theHIVE [166]	12
1.4	Eléments de la machine Blue Gene	12
1.5	<i>Virtualized Reality</i> - Synopsis de l'architecture 3D Room [110] . .	14
1.6	<i>Virtualized Reality</i> - Reconstruction d'une séquence vidéo [153] .	15
1.7	GrImage - Reconstruction d'enveloppe convexe 3D [80]	16
1.8	ViRoom - Synopsis de l'architecture	16
1.9	ViRoom - Exemple de suivi multi-caméra	17
1.10	Beobots - Vue générale et architecture [147]	17
1.11	OSSIAN - Synopsis de l'architecture	18
1.12	Reconstruction dense sur carte graphique [196]	20
2.1	Schéma d'un <i>cluster</i> classique	27
2.2	Schéma d'un <i>cluster</i> double bus	28
2.3	La machine BABYLON et deux de ses nœuds clients	29
2.4	Schéma des dispositifs d'acquisition de la machine BABYLON .	30
2.5	Architecture interne d'un nœud XServe Cluster	31
2.6	Architecture du processeur PPC 970	33
2.7	Les types de données ALTIVEC	36
2.8	Synopsis de l'algorithme de stabilisation	61
2.9	Découpage de l'image pour la détection des points d'intérêts . .	62
2.10	Mise en correspondance des primitives entre deux images à stabiliser	63
2.11	Application de la stabilisation à un flux vidéo	65
2.12	Synopsis de l'algorithme parallèle de stabilisation.	68
3.1	Déroulement d'un programme parallèle utilisant le modèle BSP [97]	88
4.1	Arbre de syntaxe abstraite de $a * b + c$	120

4.2	E.V.E. — Opérations simples sur des entiers 8 bit	122
4.3	E.V.E. — Opérations simples sur des entiers 16 bit	122
4.4	E.V.E. — Opérations simples sur des entiers 32 bit	123
4.5	E.V.E. — Opérations simples sur des réels simple précision	123
4.6	E.V.E. — Opérations composites sur des entiers 8 bit	124
4.7	E.V.E. — Opérations composites sur des entiers 16 bit	124
4.8	E.V.E. — Opérations composites sur des entiers 32 bit	125
4.9	E.V.E. — Opérations composites sur des réels simple précision . .	125
5.1	Graphe de processus communicant d'une application parallèle . .	138
5.2	Hiérarchisation d'un Graphe de Processus communiquant	138
5.3	Réécriture sous forme de squelette d'une application parallèle . .	140
5.4	Graphe de processus communicants du squelette pipeline	142
5.5	Schéma d'ordonnancement du squelette <i>farm</i>	144
5.6	Schéma d'ordonnancement du squelette <i>scm</i>	145
5.7	Structure d'une classe C++ : membres, méthodes et <i>vtable</i>	156
5.8	Structure mémorielle d'un <i>tuple</i>	165
5.9	Déroulement de l'étape de placement des tâches	171
6.1	Modèle sténopé monoculaire	183
6.2	Géométrie d'une paire stéréoscopique	184
6.3	Correction de la distorsion radiale	186
6.4	Principe de la rectification épipolaire	187
6.5	Algorithme de rectification - Plans image rectifiés [83]	190
6.6	Résultat de l'application d'un détecteur de Harris et Stephen . .	193
6.7	Stratégie d'appariement sur une droite épipolaire	194
6.8	Résultats d'une reconstruction 3D	196
6.9	Graphe de processus communiquant pour l'application de reconstruction 3D	199
6.10	Schéma d'imbrication des squelettes de l'application de reconstruction 3D	199
6.11	Approche probabiliste du suivi d'objet	206
6.12	L'algorithme CONDENSATION [100]	206
6.13	Modélisation de la position d'un piéton	207
6.14	Détecteur à ondelettes de Haar	209
6.15	Échantillon de la base d'apprentissage de piéton	209
6.16	Boite englobante d'un piéton	210
6.17	Graphe de processus communiquant pour l'application de suivi . .	214
6.18	Schéma d'imbrication des squelettes de l'application de suivi . .	215

Liste des tableaux

2.1	Comparatif Ethernet/InfiniBand/Myrinet	26
2.2	Comparaison <i>cluster</i> classique et <i>cluster</i> «double bus»	28
2.3	Evaluation des performances d'ALTIVEC [155, 52]	53
2.4	Etat du pipeline ALTIVEC sans optimisation	55
2.5	Effet d'un déroulage partiel sur le pipeline ALTIVEC	56
2.6	Taille des pipelines de l'unité ALTIVEC	56
2.7	Temps d'exécution de l'algorithme de stabilisation	65
2.8	Importance relative des étapes de la stabilisation.	66
2.9	Caractéristiques des étapes de l'application de stabilisation.	66
2.10	Performances (en <i>ms</i>) de la stabilisation — Image 640x480	69
2.11	Performances (en <i>ms</i>) de la stabilisation — Image 2560x1920	69
2.12	Performances (en <i>ms</i>) de la stabilisation — Image 5120x3840	69
2.13	Évaluation du modèle de performance hybride - Image 640x480	74
2.14	Évaluation du modèle de performance hybride - Image 2560x1920	75
2.15	Évaluation du modèle de performance hybride - Image 5120x3840	75
3.1	Comparaison des approches <i>Design Patterns</i> et Squelettes algorithmiques [53].	86
4.1	Comparaison des codes assembleurs finaux valarray/array/C	114
4.2	Effet du nombre d'opérandes sur les performances de E.V.E.	128
4.3	Temps d'exécution (ms) de RGB2YUV_C	130
4.4	Temps d'exécution de RGB2YUV_EVE	132
4.5	Temps d'exécution (ms) de RGB2YUV SIMD	133
4.6	Temps d'exécution (ms) de RGB2YUV_USIMD	133
5.1	Mesure du surcoût en débit du squelette pipeline	178
5.2	Mesure du surcoût du squelette pardo	178
5.3	Mesure du surcoût du squelette farm	178
5.4	Mesure du surcoût du squelette scm	179

6.1	Temps d'exécution de la reconstruction 3D pour 1 paire d'image sur un seul processeur	196
6.2	Importance relative des étapes de la reconstruction 3D	197
6.3	Parallélisation de l'algorithme de reconstruction 3D	198
6.4	Modélisation des gains de l'application de reconstruction 3D	201
6.5	Temps d'exécution de la reconstruction 3D en mode SIMD.	202
6.6	Temps d'exécution de la reconstruction 3D en mode SMP.	202
6.7	Accélération de la reconstruction 3D en mode MIMD.	202
6.8	Temps d'exécution de la reconstruction 3D - 500 points	203
6.9	Temps d'exécution de la reconstruction 3D - 1000 points	203
6.10	Temps d'exécution de la reconstruction 3D - 2000 points	204
6.11	Performances séquentielles du suivi de personne par filtrage particulier	212
6.12	Importance relative des étapes de l'algorithme de suivi	213
6.13	Parallélisation de l'algorithme de suivi	214
6.14	Estimation des accélérations de l'algorithme de suivi parallèle	217
6.15	Performances temporelles du suivi parallèle	220

Introduction

Depuis les débuts de l'informatique, les besoins en puissance de calcul n'ont cessé d'augmenter. Si des domaines aussi variés que la simulation météorologique, la dynamique des fluides ou la modélisation en physique des particules ont su tout d'abord utiliser au mieux les ressources des premiers ordinateurs, leurs besoins en puissance ont rapidement excédé ce que les machines séquentielles étaient capables de fournir.

La vision artificielle

Parmi les applications gourmandes en puissance de calcul, on trouve celles de la vision artificielle. La vision artificielle, ou «vision par ordinateur», est une discipline qui consiste à analyser une image ou un flux d'images afin d'en extraire des informations de haut niveau. Plus explicitement, un processus de reconnaissance [24, 75] similaire à celui opéré par l'être humain va identifier les objets contenus dans ces images par l'extraction et l'analyse de caractéristiques abstraites (comme des primitives géométriques) à partir des valeurs de chaque pixel. Les applications de la vision artificielle sont très nombreuses. On peut citer par exemple le contrôle de la qualité sur une chaîne de production, l'identification d'un individu par biométrie, le diagnostic médical, la classification de terrains à partir d'images satellites ou la commande de robots mobiles. Au regard de ces applications, on distingue deux types de contraintes.

Le volume de données traité

La quantité de données traitée par un algorithme de vision artificielle peut croître très rapidement. Si l'on considère un flux vidéo VGA cadencé à 25 images/secondes, il s'agit de traiter $3 \times 640 \times 480$ octets toutes les 40 ms, soit un débit d'entrée/sortie de 22Mo par seconde. Si le traitement consiste à appliquer 500 opérations flottantes à chaque pixel de l'image la puissance de calcul nécessaire

est de l'ordre de 10 GFLOPS; une puissance à comparer aux puissances maximales offerte par des architectures séquentielles classiques qui avoisinent les 3 GFLOPS¹.

La réactivité des applications

Les algorithmes de vision artificielle sont rarement utilisés seuls. Ils s'intègrent en général au sein d'un système qui utilise les informations qu'ils fournissent pour prendre une décision vis à vis de leur environnement proche. Ainsi, pour de nombreuses applications comme la robotique par exemple, le traitement «à la volée» des capteurs image devient une nécessité. Même si cette disposition n'implique pas forcément le traitement exhaustif de toutes les images issues des capteurs, elle soumet ce type de système à de fortes contraintes temporelles. Il devient nécessaire que l'intervalle de temps séparant l'acquisition de l'image et la réaction du système soit suffisamment court pour que l'environnement n'ait pas évolué de manière significative pour l'algorithme. Il est aussi important que ce temps de traitement soit déterministe. On se retrouve alors dans un contexte «temps réel», à savoir que l'algorithme de vision doit être exécuté à une cadence permettant de garantir la pertinence des informations générées vis à vis de l'environnement. Par exemple, une application de détection d'obstacle embarquée dans un véhicule se doit de répondre dans un intervalle de temps compatible avec l'intervalle de temps nécessaire à la réaction envisagée sur le véhicule.

Dans la plupart des cas «réalistes», ces deux facteurs – volume de données et réactivité – se combinent et augmentent d'autant la puissance de calcul nécessaire. Pour satisfaire ces besoins, plusieurs solutions sont envisageables. Les plus simples consistent soit à simplifier l'algorithme afin de limiter la quantité de calculs effectués ou à dégrader les images afin de réduire le volume de données à traiter. Ces solutions sont facilement applicables mais posent des problèmes de précision ou de robustesse qui peuvent rendre les algorithmes inutilisables. Une autre solution consiste à attendre la génération suivante de microprocesseurs. En effet, les progrès en terme d'intégration des transistors au sein des microprocesseurs ont longtemps garanti une évolution rapide de la puissance des ordinateurs. La loi de Moore [133] prévoyait en effet que, tous les 18 mois, le nombre de transistors par unité de surface au sein des processeurs doublerait. Mais deux phénomènes ont freiné cette évolution depuis le milieu des années 90 : les difficultés croissantes rencontrées par les technologies d'intégration [171] et le fait que les

¹Ces valeurs sont valables pour des machines disponibles depuis 2000-2005

technologies des autres composants annexes des machines (mémoires vives, bus etc ...) ne suivaient pas la même évolution. Ce ralentissement rend problématique le recours à des machines de type PC pour des applications de vision artificielle soumises à de fortes contraintes temporelles et demande à se tourner vers un autre type de solution.

Intérêt du parallélisme

Une de ces solutions est de faire travailler simultanément plusieurs unités de calculs et de coordonner leurs puissances pour résoudre un même problème : c'est à dire mettre en oeuvre une machine parallèle. De nombreuses applications de ce principe existent et diverses classifications de ce type de machines ont été proposées [162, 194]. Un premier niveau de classification consiste à distinguer les machines parallèles constituées de plusieurs ordinateurs reliés entre eux par un réseau de communications — on parle alors de **multi-computer** — et les machines parallèles constitué de plusieurs processeurs intégrés au sein d'une unique machine et se partageant une mémoire centrale — on parle alors de machine **multi-processeurs ou SMP**². Une classification plus synthétique est proposée par Flynn [77] et se base sur la quantification des flux de données et des flux d'instructions de la machine. On distingue alors quatre types de machine :

- **SISD** : *Single Instruction, Single Data*

Ces machines sont processeurs séquentiels conçus selon une architecture de Von Neumann [193] classique. Encore récemment, on incluait dans cette catégorie les processeurs équipant la plupart des machines personnelles. Depuis le milieu des années 90, la multiplication des unités de traitement annexes et des *pipelines* intra-processeurs au sein de ces machines a rendu floues les limites de cette classification.

- **SIMD** : *Single Instruction, Multiple Data*

Dans cette classe d'architecture, plusieurs unités de traitement exécutent simultanément un même flot d'instruction, issue d'une seule unité de contrôle sur des flots de données différents. Les machines de type CM-2 [98] datant des années 80 sont des exemples d'anciennes architectures SIMD. De nos jours, ce type de calculateur SIMD a été plus ou moins abandonné en raison de son manque de souplesse. Les techniques ainsi développées ont été remises au goût du jour dans des architectures plus flexibles. Un

²Pour *Symmetrical MultiProcessor*

exemple de ce recyclage sont les technologies de type **SWAR**³ qui ont permis le développement d'extension dites "vectorielles" ou "multimédia" comme MMX/SSE [146] d'Intel ou AltiVec [128, 142] de Motorola/IBM.

- **MISD** : *Multiple Instruction, Single Data*

Pour ces structures, les données transitent de processeur en processeur en subissant à chaque étape un traitement différent. Les calculateurs systoliques [48] sont de bons exemples de ces architectures. Ce type d'architecture reste néanmoins difficile à mettre en oeuvre et est souvent dédié à des tâches bien spécifiques.

- **MIMD** : *Multiple Instruction, Multiple Data*

Dans ces architectures, les processeurs effectuent simultanément des instructions différentes sur des données différentes. Cette forme de parallélisme est la plus répandue et a fait l'objet de nombreuses études. A titre d'exemple, on peut citer les calculateurs de type CRAY T3D [19], la CM-5 [30] ainsi que les architectures de types *cluster* (cf section 1.1).

Cette classification ne rend néanmoins pas compte de la grande diversité des machines MIMD. Pour compléter cette classification, Johnson [105] a proposé une classification des machines MIMD basée sur la structure de leur mémoire — distribuée (DM) ou partagée (GM) — et sur les mécanismes utilisés pour les communications — variables partagées (SV) ou passage de message (MP). Ces deux classifications — Flynn et Johnson — se complètent et englobent ainsi une large partie du bestiaire des machines parallèles (cf fig. 1).

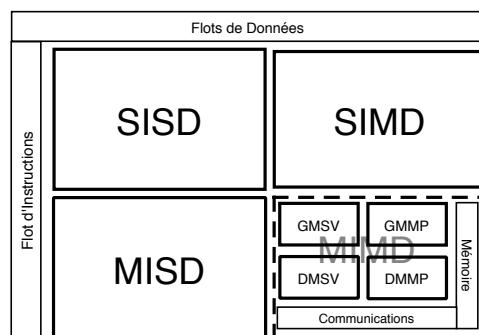


FIG. 1 – Classification de Flynn-Johnson

³SIMD Within A Register

Objectifs des travaux

Les machines parallèles peuvent répondre aux besoins de puissance de certaines applications de vision artificielle. Néanmoins, plusieurs aspects spécifiques provenant aussi bien de l'aspect matériel que de l'aspect applicatif font de l'utilisation de ce type d'architecture dans ce cadre algorithmique un sujet relativement ouvert. Parmi l'ensemble des applications de la vision artificielle, nos travaux se focalisent sur le développement et la mise en œuvre d'une architecture parallèle dédiée à l'exécution d'algorithmes de vision opérant «à la volée» d'un flux vidéo. Dans le cadre d'une telle architecture, cette classe d'algorithme met en avant deux problématiques bien distinctes.

L'architecture matérielle

La vision artificielle impose des contraintes différentes de celle des applications typiquement développées sur des machines parallèles classiques. La place prépondérante des capteurs vidéos et leur importance au sein du processus de calcul font que leur intégration au sein d'une machine parallèle nécessite d'adapter cette dernière afin de permettre un accès rapide et efficace aux données vidéos provenant de ces capteurs. En outre, les besoins de la vision artificielle nécessitent de concevoir une machine parallèle très performante.

L'environnement de développement

En sus de ces aspects architecturaux, il est important de garder à l'esprit que le développement d'applications parallèles est une tâche complexe. S'ajoute alors aux problèmes de performances une problématique de **programmabilité**. En effet, les développeurs de la communauté Vision ont à leur disposition un ensemble de modèles et de solutions logicielles éprouvés répondant aux problèmes récurrents de leur discipline. La programmation d'une machine parallèle peut s'avérer très différentes de ces modèles classiques et nécessiter une adaptation souvent problématique.

Notre objectif est donc de définir une plate-forme de développement pour cette classe d'algorithmes en fournissant un ensemble d'outils logiciels visant à faciliter le développement. Les travaux de ce manuscrit s'organisent donc de la manière suivante :

- **Chapitre 1, Architectures dédiées à la vision.** Nous présentons ici un rapide panorama des différentes architectures utilisées pour le développement et l'exécution d'applications de vision artificielle. Parmi celles ci, nous nous attarderons sur les architectures de type grappe de calcul ou *clusters*. Nous verrons en quoi cette classe d'architecture répond aux contraintes de performances mises en avant dans cette introduction et en tirerons les conclusions nécessaires à la spécification d'un *cluster* générique pour la vision artificielle.
 - **Chapitre 2, La machine BABYLON.** Nous définissons ici l'architecture de notre machine parallèle pour la vision artificielle. Cette architecture est un *cluster* dont les noeuds de calculs sont des stations de travail SMP-SIMD, exposant ainsi un degré de parallélisme plus élevé que les *clusters* classiques. Après avoir défini et argumenté nos choix technologiques, nous proposons un schéma architectural pour cette machine. Nous présentons ensuite les principaux outils disponibles pour le développement d'applications sur cette architecture que nous validons en implantant un algorithme de complexité moyenne. Cette application nous permet au passage de mettre en place un modèle de performance adapté aux architectures hybrides MIMD-SMP-SIMD et de préciser les problématiques de programmabilité spécifiques à ce type de machine parallèle.
 - **Chapitre 3, Outils logiciels le calcul parallèle** A partir des considérations du chapitre 2, nous montrons les limites des outils existants en terme de programmabilité et présentons un état de l'art des outils de plus haut niveau permettant le développement d'applications sur cette architecture. Nous montrons néanmoins que ces outils de haut niveau restent limités par leur expressivité ou leur efficacité. Parmi ces différents types d'outils disponibles, les approches semi-explicites à base de bibliothèques répondent de manière très satisfaisante aux contraintes de programmabilité. Nous proposons alors de définir deux bibliothèques C++ pour répondre à deux besoins : d'une part, la gestion efficace du parallélisme de grain fin via une bibliothèque de calcul scientifique vectoriel basée sur AltiVec; d'autre part, la gestion du parallélisme gros grain via une bibliothèque de programmation orientée squelette basée sur MPI.
 - **Chapitre 4, La bibliothèque E.V.E.** La définition d'une bibliothèque orientée objet visant à manipuler de manière efficace des tableaux numériques est un problème classique de génie logiciel. Néanmoins, nous montrons que les
-

implantations classiques de ce type de bibliothèque sont relativement peu efficaces. Après avoir identifié l'origine de cette inefficacité, nous introduisons le concept d'*évaluation partielle* et nous montrons que les techniques issues de ce concept nous permettent de s'abstraire de ces limitations en proposant un modèle de programmation haut niveau performant. Nous présentons ensuite l'interface utilisateur de E.V.E., les détails pertinents de son implantation et nous effectuons des tests de performances simples afin de valider notre concept. Enfin, nous concluons par une étude de cas réaliste qui nous permet de juger de la pertinence de l'outil.

- **Chapitre 5, La bibliothèque QUAFF** En suivant un cheminement similaire à celui de E.V.E. , nous présentons QUAFF, une bibliothèque C++ de développement d'applications parallèles à base de squelettes algorithmiques. Après avoir rappelé les principes de cette approche, nous mettons en évidence les limitations d'une implantation naïve de ce type de bibliothèque et nous montrons que les concepts d'évaluation partielle et de métaprogrammation présenté au chapitre précédent s'adaptent à cette nouvelle problématique. Nous définissons alors l'interface utilisateur de QUAFF et exposons les particularités de son implantation. Nous concluons en démontrant l'efficacité de notre approche en effectuant des tests de performances sur les squelettes proposés.
 - **Chapitre 6, Application à la stéréovision.** Nous présentons enfin l'implantation d'applications de vision artificielle. Nous nous attardons plus précisément sur les problématiques de la stéréovision et, après avoir mis en avant les spécificités de cette classe d'algorithme, nous proposons de décrire le fonctionnement de deux applications : une application de reconstruction 3D qui repose sur les caractéristiques géométriques intrinsèque de la stéréovision et une application de suivi de personne qui, en plus de ces aspects purement géométriques, met en oeuvre un algorithme probabiliste de détection. Ces applications nous permettent de montrer comment les aspects architecturaux et logiciels de BABYLON se complètent et permettent de répondre aux besoins de cette classe d'applications. Nous présentons ensuite leur implantation parallèle et évaluons leur performance afin de valider l'ensemble de la machine BABYLON.
 - **Conclusion et perspectives.** Nous résumons dans ce chapitre les résultats obtenus dans ce manuscrit. Enfin, nous discutons de quelques perspectives ouvrant de nouveaux horizons de recherche.
-

Chapitre 1

Architectures dédiées à la vision

« The hardest thing is to go to sleep at night, when there are so many urgent things needing to be done. A huge gap exists between what we know is possible with today's machines and what we have so far been able to finish»

Donald Knuth

De très nombreuses architectures ont été proposées pour résoudre les problématiques du calcul haute performance en général et de la vision artificielle en particulier. Devant la profusion des réalisations dans ce domaine [29, 145], il est bien entendu illusoire de vouloir dresser ici un panorama exhaustif. Nous allons donc nous limiter à présenter les grands types d'architectures qui sont actuellement exploitées : les *clusters*, les solutions à base de réseaux logiques programmables et celles à bases de processeurs graphiques.

1.1 Les Clusters

Pendant plusieurs années, les *super-calculateurs* s'imposèrent comme seule solution efficace aux problèmes de performances soulevés par de nombreux domaines. Néanmoins, le coût et l'infrastructure nécessaire au déploiement d'une telle machine rendaient la gestion de ces solutions relativement problématique. Dans les années 1990, l'idée d'abandonner les super-calculateurs commence à poindre avec la création de machines parallèles composées d'unités de calcul indépendantes et peu coûteuses : les *clusters*. Ces machines devinrent alors rapidement populaires du fait de leur très bon rapport performance/coût, comparées aux

super-calculateurs propriétaires et de leur facilité de mise en œuvre.

De manière synthétique, on définit un *cluster* [33] comme un système réalisant du calcul parallèle, qui consiste en un ensemble de calculateurs individuels interconnectés entre eux, et travaillant ensemble comme une unique ressource de calcul. Un noeud de calcul peut être un système monoprocesseur ou multiprocesseur comportant de la mémoire, des accès entrées/sorties, et un système d'exploitation. L'ensemble des noeuds de calcul apparaît alors comme un système unique du point de vue des applications. La figure 1.1 schématisé cette définition.

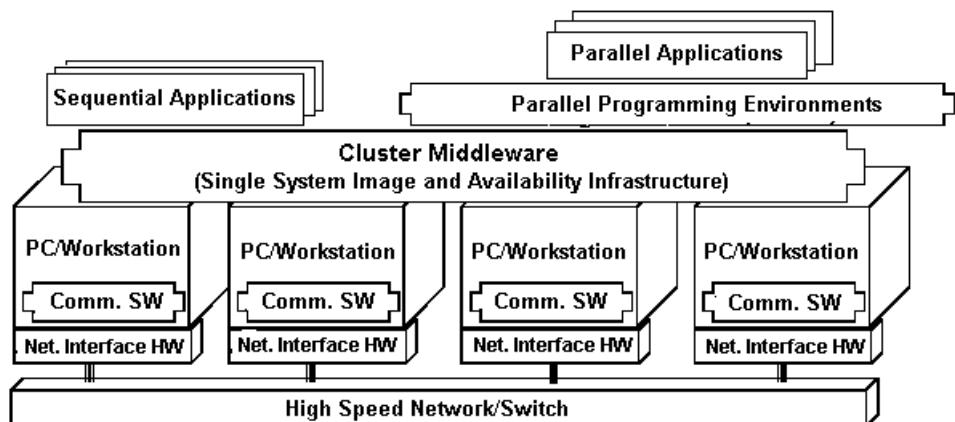


FIG. 1.1 – Architecture générique d'un *cluster* extraite de [33]

Un *cluster* est donc constitué des éléments suivants :

- Plusieurs machines individuelles (PC, station de travail ou machine SMP);
- un réseau d'interconnexion entre les différentes machines du *cluster*;
- un système d'exploitation sur chaque machine individuelle;
- un *Middleware* qui rend transparente la topologie du cluster;
- un environnement de programmation parallèle.

De très nombreux projets ont permis de mettre en avant les avantages de l'architecture *cluster*. Devant leur nombre, nous allons présenter ici les réalisations les plus marquantes historiquement et en terme de performances.

1.1.1 NOW : Networks Of Workstations

Le projet NOW (Networks Of Workstations) [15] de l’Université de Berkeley, a consisté à utiliser un grand nombre de stations de travail individuelles (quelques centaines), à les raccorder par un réseau rapide, et à y installer une bibliothèque de communication performante, afin d’obtenir des performances analogues à celles d’une machine massivement parallèle, mais avec un coût réduit. L’expérimentation NOW à Berkeley a rassemblé 100 Ultra Sparc et 40 Sparc Stations Sun sous SUN Solaris, 35 PC sous Windows NT et Unix, 300 stations de travail HP, et entre 500 et 1000 disques, le tout connecté par un réseau de commutateurs Myrinet (fig. 1.2).



FIG. 1.2 – Deux segments du *cluster* NOW

1.1.2 Beowulf

Le projet Beowulf [168, 167] a consisté en la réalisation d’une grappe de PCs sous LINUX inter-connectés par un réseau Ethernet et utilisant les protocoles de communication standards UNIX. Pour accroître les performances, plusieurs interfaces réseau sont utilisées simultanément. La principale innovation du projet Beowulf est son utilisation massive de composants standards¹, disponibles aisément.

Parmi les différentes réalisations du projet Beowulf, on trouve la machine "theHIVE"² [166] (fig.1.3). Mise en place par la NASA, "theHIVE" est un *cluster* composé de 1122 Pentium Pro bi-processeurs, 16 Pentium III Xeon bi-processeurs

¹ou *Commodity Off The Shelves*

²*the Highly-parallel Integrated Virtual Environment*

et 10 Pentium III Xeon quadri-processeurs. Ces 2316 processeurs sont répartis sur quatre sous-clusters et sont utilisés pour des applications de simulations physiques.

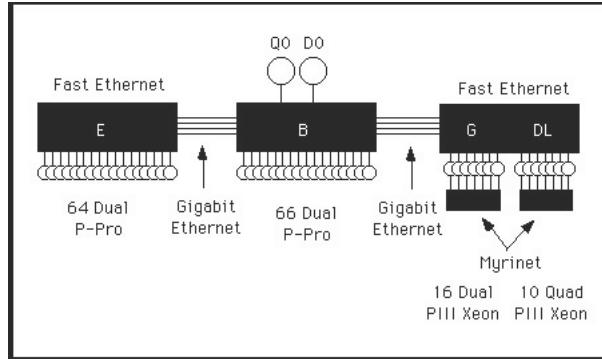


FIG. 1.3 – Architecture du *cluster* theHIVE [166]

1.1.3 Blue Gene

Blue Gene est un projet de recherche initié par IBM [176] dont l’objectif était de mettre en œuvre une machine parallèle capable d’atteindre les 360 TFLOPS et dont le domaine d’application cible englobe l’hydrodynamique, la chimie quantique, la dynamique moléculaire [76, 86], la modélisation climatique et la modélisation financière. Blue Gene est basée sur une architecture dite «cellulaire» (fig. 1.4).

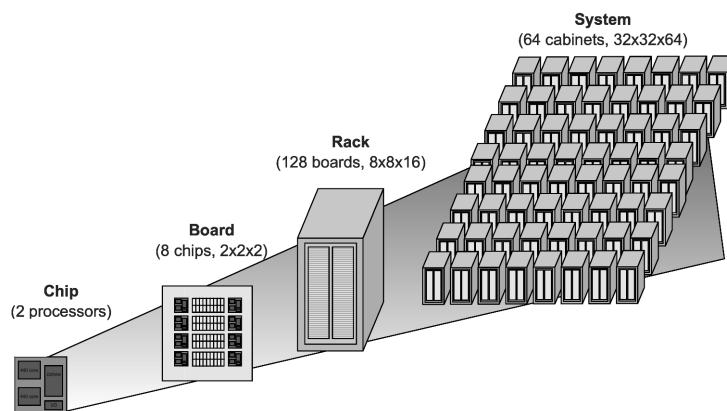


FIG. 1.4 – Eléments de la machine Blue Gene

Blue Gene est composé de 64 racks comprenant chacun 128 cartes mères. Chacune de ces cartes mères contient 8 ASIC contenant 2 processeurs IBM PowerPC cadencés à 700MHz. La machine complète contient donc 131072 processeurs reliés par plusieurs réseaux de communication :

- Un réseau dédié à la diffusion haut débit proposant une bande passante de 350 Mo/s et une latence de 1.5ms.
- Un réseau dédié aux signaux de synchronisation inter-noeuds.
- Un réseau formant un tore tridimensionnel reliant chaque noeud à ses 6 plus proches voisins et disposant d'une bande passante bidirectionnelle de 175 Mo/s.

1.2 Les Clusters pour la vision

Le passage de la recherche à l'utilisation massive des *clusters* a été rendue possible par un certain nombre de facteurs [132, 169] comme l'amélioration rapide des performances de calcul des microprocesseurs, l'amélioration comparable de la bande passante mémoire disponible et l'émergence de réseaux très haut débit comme Myrinet, InfiniBand ou Ethernet GigaBit. Les *clusters* s'imposent désormais comme une alternative viable au super-calculateurs propriétaires. Dans le domaine de la vision par ordinateur, les *clusters* se démarquent notamment des architectures comme les GPU ou les FPGA[49]. Aussi pertinentes et ciblées que soient ces solutions, les grappes de calcul présentent en effet trois avantages :

- **Rapport coût/performance** : le coût total de la machine reste très inférieure à celui d'une machine dédiée de puissance équivalente notamment grâce à l'utilisation de matériel de type *Commodity Off The Shelves*.
 - **Extensibilité et résistance à l'obsolescence** : une grappe étant construite à partir de machines individuelles interchangeables, la mise à jour d'un ou plusieurs noeuds d'une grappe reste une opération simple. De même, étendre la machine en lui ajoutant un ou plusieurs nœuds se fait facilement et à faible coût.
 - **Programmabilité** : le développement sur un cluster se fait en utilisant des outils et méthodes standards, par opposition au outils et méthodes utilisés sur des architectures comme les FPGA ou les GPU (voir section 1.3 et 1.4).
-

Ces avantages ont permis à de nombreux projets d'émerger et d'utiliser avec succès cette technologie. Les sections suivantes présentent de manière synthétique quelques travaux mettant en oeuvre une telle architecture pour l'implantation d'algorithme de vision artificielle.

1.2.1 Virtualized Reality - 3D ROOM

Le projet *Virtualized Reality* [109] mené au *Robotics Institute* de Pittsburgh sous la direction de Takeo Kanade a mis au point la *3D room*. Cette machine permet la numérisation d'une scène 3D. La *3D room* est une pièce de $6m \times 6m \times 2.7m$. contenant 49 caméras reliées à un cluster de 17 PCs (figure 1.5).

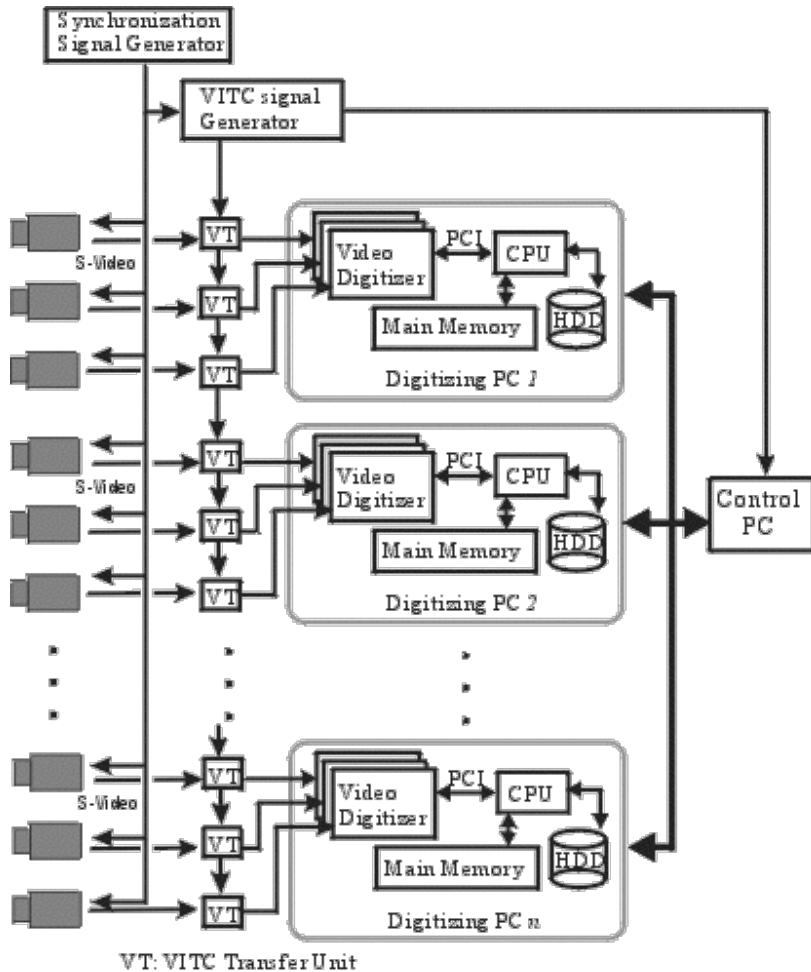


FIG. 1.5 – *Virtualized Reality* - Synopsis de l'architecture 3D Room [110]

Les 49 cameras émettent un signal synchronisé vers chaque PC capable de numériser trois flux vidéos simultanément. Chacun de ces PC exécute alors un algorithme prédéfini sur la séquence vidéo ainsi numérisée. Un PC de contrôle coordonne ensuite la mise en route, la synchronisation des acquisitions et la récupération des résultats en provenance des PC reliés aux caméras. Les applications implantées sur cette architecture [153] permettent la reconstruction 3D en temps réel de la scène filmée par l'ensemble des capteurs. Cette reconstruction peut ensuite être visionnée et permet la création de séquences vidéo utilisant des points de vues virtuels. D'autres travaux s'attachent à définir des algorithmes d'interpolations de séquence 3D [183] ou de calcul de flot optique 3D [184]. La figure 1.6 représente un extrait de séquence vidéo et sa reconstruction via la 3D Room.

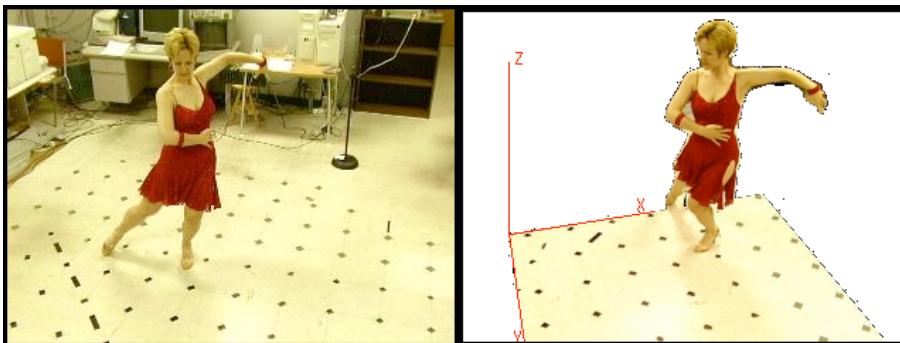


FIG. 1.6 – *Virtualized Reality* - Reconstruction d'une séquence vidéo [153]

Les éléments présentés sur cette figure sont : à gauche, la trame vidéo originale; à droite, une vue de synthèse issue de la reprojecion des textures extraites de la trame originale sur le nuage de point 3D reconstruits.

1.2.2 GrImage

L'équipe PERCEPTION de l'INRIA Rhône Alpes a mis au point la plateforme GrImage [13], dont l'objectif est de fournir un outil d'expérimentation performant permettant d'acquérir de manière synchronisée un grand nombre de flux vidéo numériques, de traiter les données sur un cluster de PC et d'afficher les résultats en haute résolution. Elle est composé d'un *cluster* de 11 PC Dual Xeon et de 16 Dual Opteron reliés par un réseau ethernet Gigabit et d'un écran d'affichage de 2m par 2.7m composé de 16 vidéo projecteurs supportant une résolution de 4096x3072 pixels. Les applications de la plate-forme GrImage incluent la reconstruction 3D temps réel [80, 131] (fig. 1.7) et la réalité augmentée [12].



FIG. 1.7 – GrImage - Reconstruction d’enveloppe convexe 3D [80]

1.2.3 ViROOM

ViROOM[172, 173] est un environnement multi-caméra mobile extensible développé par l’équipe de Tomas Svoboda et Petr Doubek pour le Laboratoire de Vision par Ordinateur de l’ETH Zurich. L’architecture de ViRoom comporte un nœud maître et une série de nœuds dédiés à l’acquisition et au traitement des données issues d’un groupe de caméras (fig. 1.8).

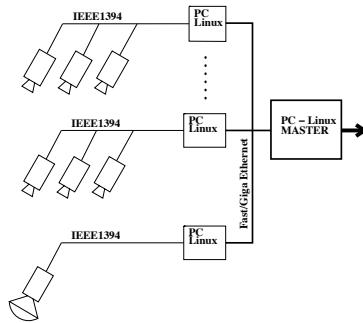


FIG. 1.8 – ViRoom - Synopsis de l’architecture

Ses applications sont centrées sur la détection et le suivi de personnes au sein d’une pièce, la reconnaissance et l’enregistrement de ces actions, la sélection du meilleur point de vue[140] et la génération d’un nouveau point de vue via une caméra virtuelle. Le système est construit de manière à ne pas être restreint à une salle particulière. Il peut être déplacé et calibré pour s’adapter à n’importe

quelle pièce. La figure 1.9 présente par exemple le résultat d'un suivi multi-caméra effectué dans une salle équipée de six caméras. La vue de droite représente la «meilleure vue» actuelle de la cible au sein du réseau de caméras. La vue de gauche représente sa position dans la salle reconstruite par ViRoom. Les six vues inférieures sont les vues respectives de chaque caméra.

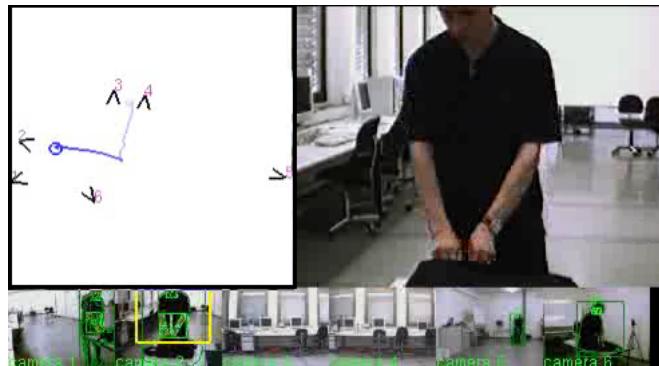


FIG. 1.9 – ViRoom - Exemple de suivi multi-caméra

1.2.4 Beobots

Le projet Beobots[136] développé au sein du laboatoire iLAB USC, vise à définir et construire des robots autonomes utilisant un *mini-cluster* permettant d'exécuter des algorithmes de neuroperception en temps réel [40]. La mise au point d'un Beobot ne nécessite que du matériel peu coûteux³.

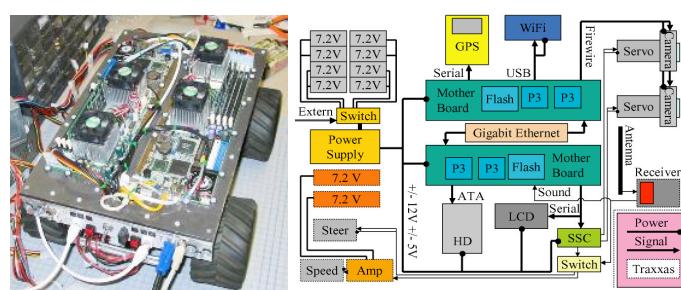


FIG. 1.10 – Beobots - Vue générale et architecture [147]

³le terme Beobots devenant la contraction de *Beowulf Robots*

L'architecture (fig. 1.10) d'un Beobot s'articule autour de deux cartes mères supportant en tout quatre processeurs Pentium III Coppermine. En outre, le Beobot est équipé d'une mémoire flash contenant un noyau Linux dédié et un câblage FireWire IEEE 1394a reliant le *cluster* aux deux caméras embarquées. Le Beobot est conçu comme une plate-forme de test pour la robustesse d'algorithme de vision neuromorphique [147]. Le but final est de permettre aux Beobots d'évoluer dans un environnement extérieur, sans contraintes ni marquage, en évitant à la volée les divers obstacles qu'il peuvent rencontrer.

1.2.5 OSSIAN

Développée au LASMEA, la machine OSSIAN est un *cluster* conçu pour exécuter des algorithmes de vision et de traitement d'images dans un contexte embarqué [149]. Cette contrainte a fortement orienté le choix des éléments du *cluster*. OSSIAN était composée de quatre Apple G4 Cube — de faible encombrement — et était pourvue de deux bus de communications : un bus ethernet dédié aux communications et un bus FireWire 1394a dédié au transfert des données vidéos[91] en provenance d'une caméra numérique (fig. 1.11). Cette technique permet de limiter les temps de communication dus aux transferts des images entre le noeud recevant les données vidéos et les noeuds appliquant un algorithme sur ces données. Les PowerPC G4 étaient en outre équipés d'une unité de calcul SIMD intraprocессeur : l'extension AltiVec[142]. Cette extension SIMD permet d'obtenir des gains allant de 2 à 16 sur des opérations bas niveau. Au final, OSSIAN pouvait fournir des gains de l'ordre de 8 à 16 avec seulement quatre noeuds.

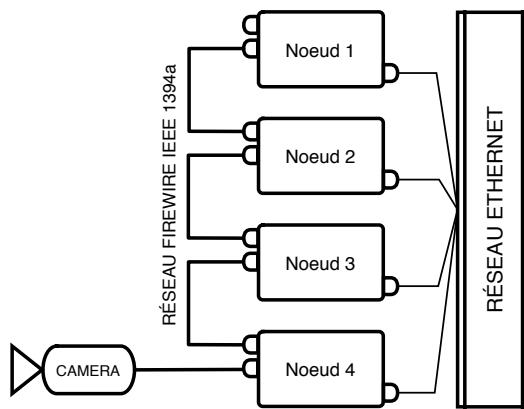


FIG. 1.11 – OSSIAN - Synopsis de l'architecture

1.3 Réseaux logiques programmables

Les FPGA⁴ sont des circuits composés de nombreuses cellules logiques élémentaires configurables et composites par l'utilisateur. Celles-ci sont connectées de manière définitive ou réversible par programmation afin de réaliser la ou les fonctions voulues. L'intérêt étant qu'un même FPGA peut-être utilisé dans de nombreux systèmes électroniques différents. Cette architecture possède de nombreux avantages qui ont permis d'y développer des applications de traitement d'images [16, 60, 59] ou de vision [28]. En effet, étant entièrement configurables, les FPGA permettent aux développeurs de spécifier une architecture en complète adéquation avec les algorithmes de vision développés. Ils permettent aussi de réaliser de manière câblée un grand nombre d'opérations. Enfin, la possibilité d'utiliser des schémas de mémoire et d'instruction divers (module DSP, communication par bus ou point à point) permet rapidement d'exploiter plusieurs niveaux de parallélisme, en particulier le parallélisme de données régulier massif.

Malheureusement, ces avancées n'ont pas permis de développer des systèmes à base de FPGA permettant d'exécuter des algorithmes de vision de très grande complexité. Bien que souples et performants, les FPGA restent limités par des soucis de programmabilité. En effet, à ce jour, la mise en oeuvre de programmes sur une plate-forme de type FPGA nécessite l'utilisation de langages de description comme VHDL, ce qui rend difficile l'utilisation généralisée de ces architectures.

1.4 Cartes d'accélération graphique

Un effet de bord de l'évolution des processeurs fut la création de processeurs dédiés au graphisme⁵ et intégrés dans des cartes spécialisées, permettant d'augmenter la puissance de calcul des machines ordinaires pour le traitement classique des sorties vidéos. Les cartes graphiques actuelles ont une puissance très élevée et peuvent être programmées [129, 141]. Du fait que ces cartes sont dédiées au traitement d'image au sens large, elles présentent un intérêt non négligeable pour l'exécution d'algorithmes de vision. L'équipe de Marc Pollefeys de l'université de Chapel Hill (Californie) a par exemple développé un algorithme de reconstruction dense d'une scène 3D [195] sur une carte graphique NVIDIA en utilisant les fonctionnalités de programmation fournies par l'API spécifique à cette carte [196].

Cette reconstruction (fig. 1.12) est effectuée à une fréquence comprise entre

⁴Field-Programmable Gate Array

⁵On parle de GPU ou *Graphical Processing Unit*



FIG. 1.12 – Reconstruction dense sur carte graphique [196]

10 et 50 images par seconde selon le nombre de niveaux de disparité et la taille des images (entre 256×256 pixels et 512×512 pixels). L'intérêt de cette approche est la grande disponibilité des cartes graphiques et leur faible coût. La plupart des cartes graphiques actuelles fournissent désormais une unité de calcul SIMD conçue spécifiquement pour les traitements bas niveau qui permet d'augmenter le débit des traitements effectués. Néanmoins, la programmabilité de ces solutions reste limitée même si elle reste supérieure à celle offerte par des solutions de type FPGA. En effet, il n'existe aucune API générique et indépendante du fabricant. En outre, pour travailler sur la sortie de l'algorithme implanté dans la carte, un processus de recopie vers la mémoire centrale est nécessaire. Ce processus peut, selon les modèles de carte et de bus graphique, devenir un goulet d'étranglement.

1.5 Conclusion

Plusieurs types d'architectures sont donc à même de satisfaire les besoins de la vision artificielle temps réel. On trouve d'une part les **FPGA** et les **GPU** qui peuvent fournir la puissance nécessaire aux traitements de type vision temps réel mais requièrent des outils et des méthodes de programmation peu flexibles et nécessitent une expertise qui fait souvent défaut aux développeurs issus de la communauté Vision. D'autre part, les *clusters* satisfont aussi la contrainte de performances mais s'appuient sur des outils plus répandus et permettent donc une transition efficace entre le développement séquentiel et parallèle.

Ceci étant, force est de constater qu'il existe peu de projets visant à utiliser un *cluster* comme architecture pour la vision artificielle. La question est donc de définir un *cluster* ouvert pouvant supporter une classe d'applications de vision

relativement large. Notre idée est donc de proposer une architecture permettant d’implanter et d’exécuter des applications de vision artificielle sans *a priori* sur la nature de l’application ou la forme du parallélisme exploité. Un projet comme GrImage [13], par contraste, repose presque entièrement sur le parallélisme de données au niveau image induit par le couplage étroit entre les capteurs et les nœuds de calcul.

Ce choix — au delà des difficultés techniques usuelles liés au déploiement d’un *cluster* — soulève néanmoins de nouveaux problèmes. Le premier de ces problèmes concerne le nombre et l’intégration des capteurs image au sein du *cluster* dont la flexibilité doit être la plus grande possible. En effet, des paramètres comme le nombre de capteurs et leurs points de connexion sur les nœuds du *cluster* doivent pouvoir être modifiés facilement pour s’adapter aux besoins des algorithmes implantés. Un problème annexe est alors de pouvoir modifier la stratégie d’acquisition — centralisée ou par multi-diffusion — en fonction de l’algorithme. En outre, il faut estimer correctement le temps nécessaire à la diffusion des informations vidéos des capteurs vers l’ensemble des noeuds de calcul. En effet, dans une optique de traitement temps réel, si le temps de diffusion devient trop important par rapport au temps de traitement d’une image, les algorithmes opérants sur cette dernière vont être fortement contraints. Une solution envisageable est alors d’utiliser un bus dédié aux communications entre les nœuds de calcul et les capteurs. Enfin, l’architecture proposée doit fournir une puissante suffisante pour répondre aux besoins des algorithmes de vision tout en conservant une taille raisonnable. Si l’on se donne comme objectif d’exécuter à la cadence vidéo — c’est à dire à une fréquence de 25 à 30 images par secondes — des algorithmes complexes dont la fréquence actuelle est de l’ordre de 0.5 à 1 image par seconde, il est nécessaire d’atteindre des gains de l’ordre de 25 à 50 et donc de déployer une machine MIMD de 50 à 100 noeuds. Ces chiffres impliquent de mettre en place une infrastructure (installation électrique et climatisation) lourde. L’utilisation d’architecture hybride MIMD-SIMD [176, 149] ou MIMD-SMP [175, 36, 99] peut *a priori* permettre d’atteindre ces gains avec un nombre de machines plus compatibles avec une infrastructure standard.

Le chapitre suivant va donc développer ces différentes solutions et présenter la mise en oeuvre d’ un *cluster* dédié à la vision : la machine BABYLON.

Chapitre 2

La machine BABYLON

*«It is a mistake to think you can solve
any major problem just with potatoes.»*
Douglas Adams

La définition d'une machine parallèle dédiée à la vision est un problème largement ouvert. Comme nous l'avons vu précédemment, parmi les types d'architectures capables de répondre aux besoins spécifiques de performance de la vision par ordinateur, les solutions à base de *cluster* ont l'avantage de s'appuyer sur des outils standards et largement diffusés dans la communauté. Mais les différents projets visant à utiliser une architecture à base de *cluster* ont conduit en pratique à des solutions *ad hoc* répondant aux besoins de classes restreintes de problèmes [109, 13] ou à des contraintes spécifiques comme l'embarquabilité [149, 136]. Il existe peu de *clusters* proposant une structure capable de supporter un large panel d'applications tout en conservant des performances permettant l'exécution à une cadence élevée de ces applications. Nous proposons donc de définir une architecture de *cluster* qui va nous permettre de répondre à ces diverses contraintes de performances et de flexibilité. Nous mettrons ensuite ce modèle architectural en oeuvre au sein de notre *cluster* — la machine BABYLON¹ — en décrivant les différents choix technologiques effectués. Nous nous attarderons ensuite sur l'architecture logicielle d'une telle machine. Nous validerons ensuite notre approche en proposant l'implantation d'une application de vision artificielle de complexité moyenne et en proposant un modèle de performance adapté.

¹La ville de Babylone est connue pour avoir hébergé en son sein la mythique Tour de Babel. Au delà du mythe biblique du danger de vouloir se placer à l'égal des dieux, elle se fait l'écho de la nécessité qu'a l'humanité de se rapprocher et de se comprendre pour réaliser de grands projets.

2.1 Architecture matérielle

La définition de notre architecture de *cluster* pour la vision nécessite de choisir de manière pertinente les différents éléments qui le composent. Plus particulièrement, nous nous attachons dans les sections suivantes au choix des nœuds de calcul, du réseau de communication et de sa topologie.

2.1.1 Noeud de calcul

La puissance nécessaire pour répondre aux besoins des algorithmes de vision augmente rapidement et le déploiement d'une machine MIMD classique demanderait l'utilisation de très nombreux noeuds pour obtenir un gain non négligeable. Dans notre contexte, une telle machine pose rapidement des problèmes d'infrastructure — principalement concernant la climatisation de la machine et la gestion de la connectique — et de consommation électrique. Il est donc nécessaire d'utiliser des noeuds de calcul proposant un ou plusieurs niveaux de parallélisme supplémentaires. Un autre point à considérer est que de nombreux algorithmes de vision reposent sur un pré-traitement des données images. Ces pré-traitements sont en général des opérations régulières, iconiques et de bas niveau et qui sont donc susceptibles d'être grandement accélérées par une implantation SIMD [58, 146].

Deux types de technologies sont alors envisageables pour fournir un nœud de calcul performant dans le cadre d'application de vision artificielle. La première est d'utiliser des noeuds possédant plus d'un processeur [175, 117], la deuxième est d'utiliser des processeurs possédant une unité de calcul SIMD interne [176, 149]. Ces technologies hybrides mettant en oeuvre plusieurs niveaux de parallélisme (MIMD-SIMD, MIMD-SMP voire MIMD-SMP-SIMD) permettent alors d'obtenir des gains élevés tout en limitant la taille de l'infrastructure.

2.1.2 Réseau de communication

Les performances d'un *cluster* dépendent aussi de la qualité de son réseau de communication. Deux problèmes se posent et concernent respectivement la technologie du réseau en elle même et sa topologie. De nombreuses technologies [179, 180, 123] sont communément utilisées dans le domaine du calcul haute-performance, en particulier :

- **Ethernet.**

Ethernet est le résultat des recherches effectué par Xerox depuis le début

des années 70 et s'est imposé comme le standard le plus utilisé, évoluant des premières versions à 10 Mb/s jusqu'à la création de l'Ethernet Gigabit, standardisé en 1998 par l'IEEE (IEEE 802.3e et 802.3ab). Ce standard définit les protocoles de transmission Ethernet sur des paires croisées de catégorie 5 ou 6 — On parle alors de Ethernet 1000BASE-T — et a permis à l'Ethernet Gigabit de s'imposer comme la technologie réseau dominante sur les machines personnelles. L'Ethernet Gigabit permet de fournir des débit proches de 1 Gb/s avec une latence de $6\mu s$. Son principal avantage réside dans son faible coût et son large support dans la communauté réseau.

- **InfiniBand.**

Fruit de la fusion de deux technologies concurrentes, Future I/O, développée par Compaq, IBM, et Hewlett-Packard, et Next Generation I/O, développée par Intel, Microsoft, et Sun Microsystems, la spécification InfiniBand prévoit une modification radicale de l'architecture traditionnelle d'entrées-sorties des serveurs informatiques. Ce standard s'appuie en effet, non sur un concept de bus partagé (comme PCI ou PCI-X), mais sur une matrice de commutation reliant par le biais de «liens» point à point à très haut débit les sous-systèmes d'entrées/sorties comme le SCSI, l'Ethernet, la mémoire principale et les processeurs. Bien que conçu comme une liaison série, InfiniBand fournit un débit typique de 2,5 Gb/s et peut atteindre jusqu'à 5 ou 10Gb/s pour une latence $6\mu s$. Les liens InfiniBand peuvent être reliés par groupe de 4 ou 12, permettant ainsi d'atteindre des débits de 120 Gb/s. Classiquement, les systèmes à base d'Infiniband utilisent des groupes de 4 liens. Dans le cas des *clusters*, plusieurs groupes de 12 liens sont agrégé via des commutateurs. InfiniBand utilise une topologie dite de "switched fabric" dans laquelle plusieurs machines peuvent se partager le réseau de manière simultanée, contrairement à une topologie en bus. Les données sont transmises par paquets de 4Ko et peuvent utiliser soit des accès mémoires directs de noeuds à noeuds, des opérations à base de transaction, des envois par canaux réservé ou de la multi-diffusion. La technologie InfiniBand reste néanmoins peu utilisée car faiblement standardisée et extrêmement coûteuse.

- **Myrinet.**

Myrinet est un système de connexion réseau à grande vitesse conçu par la firme Myricom comme une solution efficace aux problématiques de connexion au sein des *clusters*. La latence induite par Myrinet sur les temps de communication est très faible de par la simplicité du protocole utilisé et permet

d'obtenir des débits proches de la limite théorique maximale de la couche physique de transmission — soit près de 10Gb/s pour une latence de $3\mu s$. Cette couche physique est composée de deux câbles en fibre optiques — un pour les transmissions descendantes, un pour les transmissions montantes — connectées à la machine hôte par un connecteur adapté. Les différentes machines sont alors reliées via un *switch* dédié. En novembre 2005, 20% des clusters du CLUSTER TOP 500 [132] utilisaient une connexion de type Myrinet, faisant de cette dernière l'une des plus répandues dans le domaine du calcul haute-performance. Le principal défaut de Myrinet reste son coût prohibitif pour de petites structures.

Le tableau 2.1 présente un comparatif des performances de ces diverses technologies appliquées de manière typique au sein d'un *cluster*.

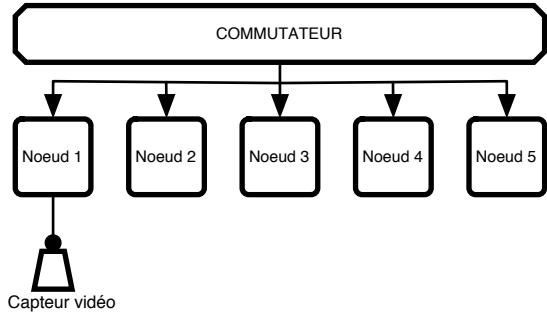
Technologie	Débit Max	Latence
Ethernet GB	1 Gb/s	$65\mu s$
InfiniBand	10 Gb/s	$6\mu s$
Myrinet	10 Gb/s	$3\mu s$

TAB. 2.1 – Comparatif Ethernet/InfiniBand/Myrinet

Globalement, InfiniBand propose le meilleur débit et la latence la plus faible. Il reste néanmoins peu standardisé et coûteux à déployer. Myrinet et l'Ethernet Gigabit ont des performances relativement proches en terme de débit mais Myrinet s'impose par sa latence très faible.

2.1.3 Topologie des réseaux de communications

Un autre point important concerne l'adaptation de la technologie du *cluster* aux problématiques de la vision artificielle. Au sein d'un *cluster* classique, le réseau de communication principal est souvent constitué de connections point à point. Ces connections sont effectuées via un commutateur (ou *switch*) qui permet à chaque nœud de communiquer avec n'importe quel autre nœud (fig. 2.1). Dans le cas d'un *cluster* dédié à la vision, l'acquisition des données images est souvent déléguée à un nœud directement connecté au capteur image. Une fois les données acquises, ce nœud procède à une diffusion de ces dernières à l'ensemble des noeuds du cluster. Plusieurs stratégies de diffusion existent et dépendent de l'implantation des primitives de communication au sein du *Middleware* utilisé.

FIG. 2.1 – Schéma d'un *cluster* classique

Dans le cas le plus défavorable, la latence λ_{dispo} , c'est-à-dire le temps nécessaire pour que chaque nœud ait accès aux données images est donné par :

$$\lambda_{dispo} = \tau_{acq} + P \cdot \tau_{diff}$$

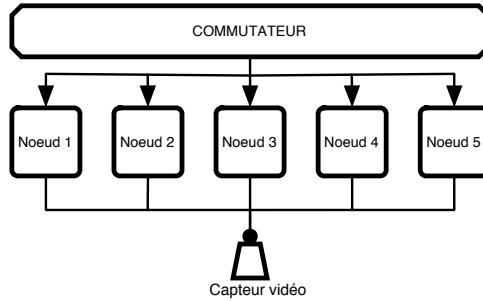
où τ_{acq} représente le temps nécessaire à l'acquisition des données sur le nœud Maître, P le nombre de nœud réclamant des données et τ_{diff} le temps de diffusion des données d'un nœud à l'autre. Dans cette configuration, la latence peut devenir très importante. En pratique, λ_{dispo} peut représenter une fraction non négligeable du temps séparant deux acquisitions vidéo. Plus grave, il augmente lorsque le nombre de processeurs P augmente, limitant de facto l'extensibilité de l'architecture.

Une solution consiste donc à fournir un mécanisme de diffusion dédié aux données vidéo. Ce mécanisme, agissant comme un réseau parallèle au réseau de communication classique doit permettre à chaque nœud de récupérer la totalité des données vidéo en provenance des capteurs sans avoir à attendre une communication provenant d'un unique nœud dédié à l'acquisition (figure 2.2). Les étapes de diffusion des données entre les capteurs et les nœuds de calcul sont alors remplacées par une simple étape de synchronisation suivie d'une acquisition en provenance directe du ou des capteurs images. Au final, la latence est donnée par :

$$\lambda_{dispo} = \tau_{synchro} + \tau_{acq}$$

où $\tau_{synchro}$ représente le temps nécessaire à synchroniser l'ensemble des nœuds via le réseau de communication principal et τ_{acq} le temps nécessaire pour effectuer une acquisition sur un nœud.

Le tableau 2.2 présente les temps de synchronisation et de transfert mesurés sur un *cluster* équipé d'un réseau Ethernet GigaBit et d'un réseau FireWire. Dans

FIG. 2.2 – Schéma d'un *cluster* double bus

ces tests, nous transmettons une image 640x480 en niveaux de gris sur 2 à 12 processeurs. Les temps présentés comprennent : le temps de synchronisation, le temps de diffusion et le temps d'acquisition depuis une caméra FireWire. Nous évaluons alors la latence dans le cas d'un *cluster* classique (λ_{simple}) et dans celui d'un *cluster* disposant d'un bus de diffusion dédié (λ_{dual}).

Noeuds	$\tau_{synchro}$	τ_{diff}	τ_{acq}	λ_{simple}	λ_{dual}
2	0.28 ms	6.98 ms	0.38 ms	7.06 ms	0.66 ms
4	0.41 ms	11.48 ms	0.38 ms	11.86 ms	0.79 ms
6	0.70 ms	19.17 ms	0.38 ms	19.55 ms	1.08 ms
8	1.01 ms	27.53 ms	0.38 ms	27.91 ms	1.39 ms
10	1.03 ms	28.53 ms	0.38 ms	28.91 ms	1.51 ms
12	1.07 ms	32.23 ms	0.38 ms	32.61 ms	1.54 ms

TAB. 2.2 – Comparaison *cluster* classique et *cluster* «double bus»

Très vite, il apparaît que la latence devient prohibitive dans le cas du *cluster* classique. Ainsi, si l'on exécute un algorithme nécessitant une telle diffusion à la cadence de 25 images par secondes, le temps nécessaire à la diffusion des données sur un *cluster* de 12 nœuds représente 32 des 40 ms disponibles ce qui impose alors de très fortes contraintes sur le temps d'exécution effectif d'un tel algorithme. Dans le cas d'un *cluster* disposant de deux bus, la latence de diffusion reste inférieure à 2 ms. L'intérêt d'utiliser à la fois un réseau classique pour le transfert des messages et un réseau dédié à la diffusion vidéo est alors flagrant. L'efficacité de cette solution a d'ailleurs été démontré par les travaux sur la machine OSSIAN [149] et par les travaux de Yoshimoto et Arita [91].

2.2 Synopsis général

A partir de l'ensemble des considérations concernant le type de nœud de calcul et le réseau de communication, nous avons défini l'architecture d'un *cluster* dédié à la vision [72] qui met en oeuvre :

- Une architecture hybride à trois niveaux de parallélisme : SIMD intégré au processeur, SMP inter-processeurs et MIMD inter-nœuds. Ce type d'architecture permet d'augmenter de manière significative le rapport performance/coût d'une telle machine. Nous avons choisi d'utiliser des nœuds de calcul qui permettent d'exploiter plusieurs degrés de parallélisme. Compte tenu de l'expérience acquise lors du développement de la machine OSSIAN et de l'évolution des machines Apple, notre choix s'est porté sur le XServe POWER PC G5. BABYLON (fig. 2.3) est donc constituée de quinze machines de ce type : douze nœuds de type XServe POWER PC G5 et trois nœuds POWER PC G5 Desktop. Le système d'exploitation utilisé est MAC OS X car les outils et drivers disponibles pour l'administration système et les accès bas niveau sont largement plus développés que sous un système Linux.



FIG. 2.3 – La machine BABYLON et deux de ses nœuds clients

- Un double réseau de communication. Ce double réseau permet de limiter l'impact du transfert vidéo entre chaque nœud et donc d'augmenter le temps processeur effectivement disponible pour les calculs. Le premier réseau de communication est constitué d'un réseau Ethernet Gigabit qui relie l'ensemble des nœuds de BABYLON à travers un commutateur. Ce réseau est entièrement dédié à la synchronisation des nœuds, l'échange de messages spécifiques à l'application et à la récupération des résultats. Le deuxième

réseau est lui entièrement dédié au transfert des images en provenance des caméras vers les nœuds de calcul et est constitué d'un réseau FireWire IEEE 1394a à 400Mo/s entre BABYLON et ses nœuds clients et d'un réseau FireWire IEEE 1394b à 800Mo/s entre ses nœuds de calcul. Dans notre cas, le faible coût de l'Ethernet Gigabit ainsi que son support natif sur les machines de types POWER PC G5 l'ont emporté sur les considérations de performances des solutions comme Myrinet ou Infiniband.

- Un système d'acquisition vidéo adaptable à la fois à des applications monoculaires et stéréoscopiques. Le capteur principale est une paire de caméras Firewire IEEE 1394a. Cette paire stéréoscopique calibrée est fixée à un support mural et est utilisable au sein d'applications de vision stéréoscopique (fig 2.4). BABYLON supporte aussi jusqu'à deux caméra FireWire supplémentaires. Ces caméras peuvent être raccordées au réseau de diffusion et être déplacées à volonté dans le cadre d'applications ne nécessitant pas de caméras inté-calibrées.

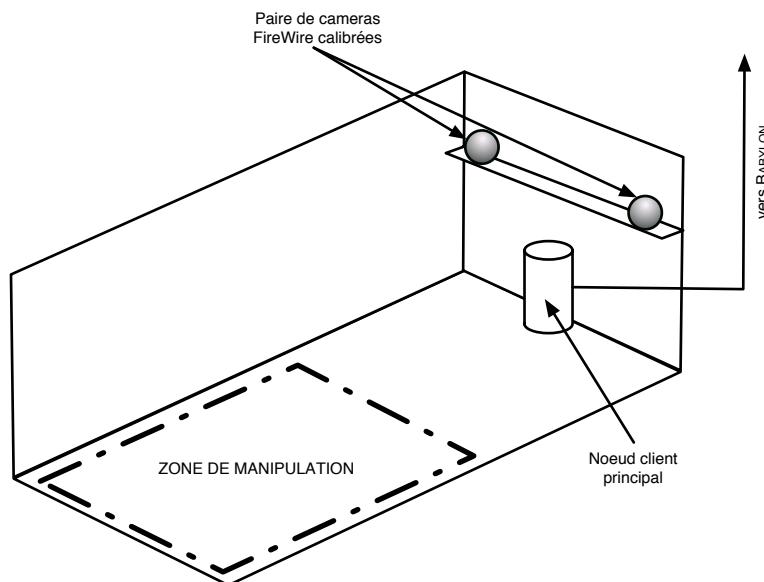


FIG. 2.4 – Schéma des dispositifs d'acquisition de la machine BABYLON

Les sections suivantes vont s'attacher à présenter les caractéristiques des nœuds XServe POWER PC G5 composant BABYLON et les caractéristiques de leur processeurs, le PPC 970.

2.2.1 Architecture du POWER PC G5

L'architecture interne du POWER PC G5 (fig. 2.5) met en oeuvre les éléments suivants :

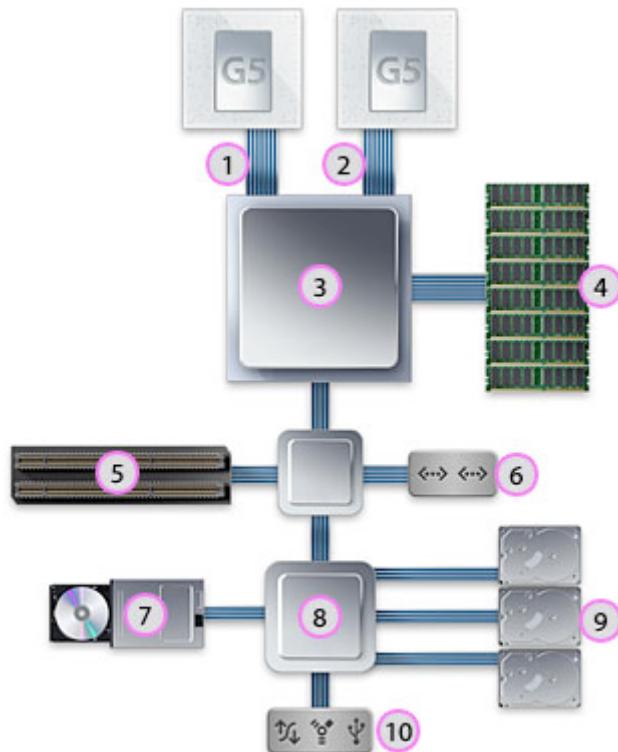


FIG. 2.5 – Architecture interne d'un nœud XServe Cluster

- **Deux processeurs PPC 970** qui seront décrits dans la section suivante.
- **Deux bus bidirectionnels indépendants** (1, 2) cadencés à 1,15 GHz. Ces bus 64 bits à double débit optimisent les transferts entre chaque processeur et le contrôleur système (3). Ces bus intègrent deux chemins de données internes haut débit unidirectionnels de 32 bits; l'un achemine les données en continu à destination du processeur et l'autre en provenance du processeur. Cela permet aux données de circuler simultanément dans les deux directions, sans temps d'attente lorsque le processeur et le système négocient l'utilisation du bus. En outre, les flux de données incluent également des signaux d'horloge, ce qui permet au bus frontal de fonctionner jusqu'à 1,15 GHz pour offrir des débits pouvant atteindre 9,2 Go/seconde. Chaque

processeur bénéficie d'une interface bidirectionnelle dédiée avec le contrôleur système, ce qui permet d'atteindre 18,4 Go/seconde de bande passante totale. Outre cet accès accéléré à la mémoire principale, l'architecture de bus frontal hautes performances permet à chaque processeur PPC 970 de détecter et d'accéder aux données présentes dans les caches L1 et L2 de l'autre processeur.

- **Un contrôleur système** (3) qui joue un rôle central au regard des performances globales du POWER PC G5 en offrant à chaque processeur un accès réservé à la mémoire centrale, évitant par là les conflits d'utilisation de la bande passante à la différence des systèmes qui partagent un bus et se disputent en permanence l'accès et la bande passante sur une voie de données commune. De plus, le contrôleur système permet aux processeurs d'adresser deux blocs de SDRAM (4) simultanément, en lisant et en écrivant à la fois sur les fronts montant et descendant de chaque cycle d'horloge, doublant la bande passante effective et permettant au POWER PC G5 d'atteindre un débit mémoire maximal de 6,4 Go par seconde via un accès direct à la mémoire (DMA).
 - **Deux interfaces Ethernet Gigabit intégrées** (6) supportées par des circuits spécialisés (ASIC) intégrées à la carte mère principale. Ces ASIC incluent deux interfaces Ethernet 10/100/1000BASE-T indépendantes, chacune avec sa propre interruption, sur un bus PCI-X 64 bits à 133 MHz dédié. La puce Ethernet Gigabit utilise des sommes de contrôle TCP, IP et UDP générées par matériel pour détecter les éventuelles corruptions de paquets et erreurs de transmission. De plus, un tampon de 64 Ko prend en charge les trames étendues, ou les paquets jusqu'à 9 Ko, pour réduire la charge système et optimiser le débit des activités réseau. Ces interfaces nous permettent d'obtenir des performances très satisfaisantes en terme de débit et de latence au sein de BABYLON .
 - **Plusieurs périphériques d'entrées-sorties** comme un bus d'extension PCI-X (5), un lecteur optique (7), des ports FireWire, USB 2.0 (10) et les ports série , des disques de stockage Serial ATA (9). L'ensemble des ces périphériques est intégré via une interconnexion HyperTransport 600 MHz (8), pour un débit maximal de 1,6 Go/s. En particulier, deux ports FireWire 800 sur le panneau arrière et un port FireWire 400 en façade assurent la connexion avec les périphériques FireWire IEEE 1394 nécessaire à la constitution de notre réseau de diffusion vidéo.
-

2.2.2 Spécification du processeur PPC 970

Chaque POWER PC G5 dispose de deux processeurs PPC 970 constitué chacun des éléments suivants (fig. 2.6) :

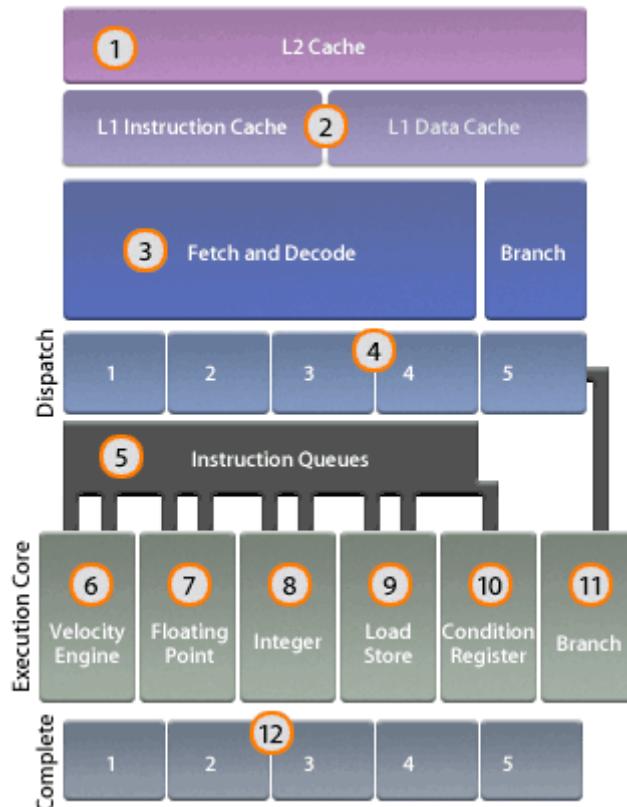


FIG. 2.6 – Architecture du processeur PPC 970

- **Deux caches hautes performance** (1,2) qui fournissent respectivement 512Ko (pour le cache L2) et 64Ko (pour le cache L1) en accès direct à 64 Mo/s et 64 Go/s.
- **Une unité d'acheminement et de décodage** (3) qui achemine jusqu'à huit instructions par cycle d'horloge du cache L1, les décode et les partage en opérations plus petites et plus simples à organiser. Cette préparation efficace optimise la vitesse de traitement alors que les instructions sont réparties dans le noyau d'exécution et que les données sont chargées dans les nombreux registres des unités fonctionnelles.

- **Une unité de répartition** (4) qui réorganise les instructions par groupes de cinq et les répartit dans les unités fonctionnelles. Le PPC 970 peut ainsi suivre jusqu'à 20 groupes en même temps aux différents stades d'acheminement, de décodage et de mise en file d'attente. Cette répartition des données est plus tard réorganisée par **l'unité de complétion** (12).
 - **Des files d'attentes** (5) associées à chaque unité fonctionnelle. Au sein de ces files, chaque groupe d'instructions en provenance de l'unité de répartition est séparé en instructions individuelles, qui passent ensuite à l'unité fonctionnelle correspondante.
 - **Une unité de calcul SIMD** (6) qui sera décrite en détail dans la section suivante.
 - **Deux unités en virgule flottante à double précision** (7) qui fournissent la précision nécessaire pour les calculs scientifiques complexes. Contrairement aux processeurs 32 bits qui exécutent des calculs 64 bits à double précision en plusieurs cycles d'horloge, le PPC 970 est capable, grâce à son architecture 64 bits et à ses deux unités de calcul pipelinées, de fournir deux résultats par cycle d'horloge.
 - **Deux unités de calcul entiers** (8) capables d'exécuter un large éventail d'instructions impliquant à la fois des calculs 32 bits et 64 bits. Elles tirent aussi parti des registres et des chemins de données 64 bits pour réaliser des calculs 64 bits en nombre entier en une seule passe.
 - **Deux unités de chargement/rangement** (9) qui chargent les données du cache L1 vers les registres de données des unités qui traiteront les données lorsque les instructions entrent dans la file d'attente. Une fois les données manipulées par les instructions, ces unités les stockent à nouveau sur le cache L1, L2 ou dans la mémoire principale. Chaque unité fonctionnelle est équipée de 32 registres de 128 bits sur l'extension ALTIVEC et de 64 bits sur les unités en virgule flottante et sur les unités en nombre entier. Ces deux unités permettent au PPC 970 de remplir ces registres de données en permanence afin de fournir une efficacité de traitement optimale.
 - **Un registre des conditions** (10) qui récapitule l'état des unités en virgule flottante et en nombre entier. Le registre des conditions indique aussi les résultats des comparaisons effectuées et fournit un moyen de les tester comme
-

conditions de branchement.

- **Une unité de prédition** (11) qui permet d'anticiper le déroulement du flot d'instructions en exécutant de manière spéculative une ou plusieurs instructions avant même qu'elles n'aient besoin d'être évaluées. Si la prédition est correcte, le processeur fonctionne plus efficacement — puisque l'opération spéculative a exécuté une instruction avant qu'elle ne soit requise. Si la prédition est erronée, le processeur doit effacer l'instruction et les données associées inutiles, créant ainsi un espace vide appelé «bulle de pipeline». Les bulles de pipeline réduisent les performances car le processeur marque un temps d'arrêt en attendant l'instruction suivante. Le G5 peut prédire l'étape suivante avec une précision pouvant atteindre 95%, permettant au processeur d'utiliser efficacement chaque cycle de traitement.

L'intérêt principal de ce processeur réside dans les multiples unités de traitements qui permettent d'effectuer un grand nombre de calculs différents simultanément. Outre ces différents niveaux de parallélisations au sein de ce processeur, l'extension ALTIVec est un élément prépondérant de cette architecture.

2.2.3 Spécifications de l'extension SIMD ALTIVec

L'origine d'ALTIVec se situe au milieu des années 90, lorsque différents fabricants de micro-processeurs s'intéressèrent au développement d'unités de calculs parallèles qui permettraient d'alléger la charge du processeur en traitant de front de grandes quantités de données. En effet, l'essor des activités multimédia avait mis en évidence les besoins en puissance de ces applications par rapport à des applications classiques comme la bureautique. Les premiers essais de HP et de Sun MicroSystem [165] permirent de définir les bases des unités de traitement SIMD dans le cadre de telles applications et ouvrirent la voie à Intel et à Motorola. Ainsi, en 1996, Intel finalisa son extension MMX [146] qui permettait de travailler sur des données 64 bits via une unité de calcul vectoriel. Cette extension se développa pour donner naissance en 1998-2003 aux unités SSE, SSE2 et SSE3 qui, en reprenant la base de MMX, autorisent le travail sur des flottants simple ou double précision dans le cadre d'applications multimédia comme la compression vidéo ou la synthèse d'images 3D.

Pourtant, malgré l'avancée que représentait MMX et de ses successeurs en termes de performances, cette technologie semblait être sous-exploitée dans les communautés de développeurs. En effet, des problèmes comme la non-orthogonalité

de leurs jeux d'instructions, les problèmes d'alignement mémoires et les problèmes d'adéquations entre les structures de données et les limitations de ces unités ralentissaient grandement le développement d'application de grande envergure. Parallèlement aux travaux d'Intel et en constatant les problèmes auxquels les fabricants de microprocesseurs se heurtaient, Motorola s'allia en 1996 à Apple pour concevoir la future architecture du Power PC G4. Keith Diefendorff, alors chef du projet ALTIVec, définit les grandes lignes de cette future unité de calcul en n'en faisant pas qu'une simple extension multimédia, mais plutôt une unité de calcul polyvalente. Tirant partie des expériences de ses concurrents, Motorola décida donc de reprendre à zéro la conception matérielle de ce type d'unité de calcul. Finalement, en 1999, le PowerPC G4 d'Apple se vit doté de l'unité de calcul ALTIVec de Motorola [58]. En 2002, IBM reprend le projet ALTIVec est l'intègre dans les processeurs PPC 970 qui équipent les machines POWER PC G5 .

ALTIVec est basé sur une implémentation intra-processeur du modèle SIMD. Dans ce modèle, la manipulation de structures de données spéciales permet de reproduire le comportement d'une unité SIMD au sein d'un seul processeur [151, 58]. Ces types de données nommés *vecteurs* sont des blocs de 128 bits pouvant contenir des données de plusieurs types (cf. figure 2.7) et sont traitées par des opérateurs spécifiques qui effectuent leurs calculs simultanément sur tous les éléments du vecteur.

1.		16 Entiers 8 bits
2.		8 Entiers 16 bits
3.		4 Entiers 32 bits
4.		4 Flottants 32 bits
5.		8 Pixels RGBA 16 bits
6.		4 pixels RGBA 32 bits

FIG. 2.7 – Les types de données ALTIVec

Au sein du PPC 970 , l'unité ALTIVec, est complètement indépendante des unités de calcul classiques. Elle possède quatre unités de traitement équipées de

pipelines qui lui permettent de traiter plusieurs opérations sur des flottants simple précision, des opérations de permutations, des opérations simples sur des entiers (comme l'addition) ou des opérations composites sur des entiers (comme les multiplications partielles) de manière efficace. ALTIVEC est en outre capable de démarrer simultanément l'exécution de plusieurs opérations dans plusieurs unités de calcul — typiquement quatre instructions pour le PPC 970 , on parle alors d'unité super-scalaire de degré 4. L'utilisateur a en outre la possibilité d'utiliser l'ensemble des registres scalaires fournis par le PPC 970 et les 32 registres vectoriels fournis par ALTIVEC de manière simultanée. Nous verrons dans la section 2.6 comment cette structure influe sur l'écriture effective de code ALTIVEC efficace et quelles solutions sont proposées pour tirer partie du maximum de la puissance fournie par cette extension.

L'ensemble de ces éléments permet au POWER PC G5 de fournir une puissance de calcul élevé de l'ordre de 8 GFLOPS. Le XServe POWER PC G5 présente donc les caractéristiques requises pour un nœud de calcul pour un *cluster* hybride exposant à la fois une nature MIMD, SMP et SIMD. Un exemple de son utilisation au sein d'un *cluster* est le projet System X mené par l'équipe *Virginia Tech*. Ce *cluster* met en oeuvre 1100 XServe POWER PC G5 pour atteindre une puissance de 12,25 TFLOPS [150].

2.2.4 Conclusion

Au-delà des spécifications matérielles de BABYLON, il est important de garder à l'esprit l'ensemble des problématiques de programmabilité. En effet, il nous semble nécessaire qu'une architecture comme BABYLON soit exploitable par les membres de la communauté vision. D'une manière générale, la programmation d'une telle machine, dans le contexte applicatif de la vision artificielle, pose en effet des problèmes théoriques et techniques complexes. Ces problèmes proviennent notamment de la diversité des modèles, langages et outils de programmation utilisés — en particulier pour exploiter le parallélisme offert par une architecture incorporant plusieurs niveaux de parallélisme — et de la difficulté — compte tenu de la complexité du processus de développement et de mise au point — à évaluer les performances d'un nombre significatif d'implantations d'un même algorithme.

Il va donc être important de bien identifier les aspects de l'architecture logicielle d'un tel *cluster* afin d'estimer les éventuelles difficultés qui se poseront aux utilisateurs peu familiers de ces modèles et outils de programmation. Le développement d'applications sur BABYLON nécessite la coordination de trois niveaux

de parallélisme utilisé deux à deux : le niveau SIMD et le multiprocesseur d'une part et les communications entre multiprocesseurs d'autre part. On doit en outre faire appel à une bibliothèque dédiée à l'acquisition de donnée en provenance des caméras FireWire. Pour ce faire, nous utilisons la bibliothèque MPI [79] pour la gestion du parallélisme de type MIMD, l'extension ALTIVec pour la programmation SIMD et la bibliothèque C+FOX pour l'acquisition des données vidéos. La gestion du parallélisme SMP est un problème plus ouvert. En effet, plusieurs technologies sont disponibles pour permettre de développer des applications tirant parti de ce type de parallélisme. Nous proposons donc de présenter l'ensemble de ces outils et de définir leurs apports en terme de performance et leurs limites respectives en terme de programmabilité.

2.3 La bibliothèque d'acquisition vidéo C+FOX

L'acquisition de données vidéo en provenance d'un bus FireWire sous Mac OS X est classiquement effectué en utilisant l'API de développement fournie par *Quicktime*. Malheureusement, cette API ne supporte ni la multi-diffusion ni la synchronisation entre les différents clients FireWire. Enfin, la fréquence d'acquisition de cette bibliothèque ne peut dépasser 25 images par secondes quel que soit le modèle de la caméra.

Afin de combler ces manques, nous avons développé une bibliothèque d'acquisition basée sur la couche bas niveau de gestion des périphériques FireWire fournies par Apple. Cette bibliothèque — appelée C+FOX² [66] — permet d'abstraire l'ensemble des tâches nécessaire à l'acquisition vidéo d'une ou plusieurs caméras FireWire en utilisant un modèle de programmation basé sur les flux d'entrée/sortie C++ .

2.3.1 Interface utilisateur

L'utilisation de C+FOX se fait comme illustré sur le listing 2.1. Dans cet exemple, nous initialisons une caméra recevant ses données du canal 0 au format 640×480 à la fréquence de 30 images par seconde. C+FOX fournit un support pour l'ensemble des modes d'acquisition des caméras compatibles FireWire IEEE 1394. La boucle principale utilise ensuite la surcharge de l'opérateur d'extraction de flux sur l'instance de Camera et permet l'accès aux dernières données images.

²C++ Camera For OS X

Cette zone mémoire est entièrement gérée par C+FOX et ne nécessite pas de copie supplémentaire tant qu'aucune modification n'y est apportée. Directement modifiée par un circuit DMA dédié, cette zone mémoire est mise à jour de manière efficace par un système de *triple buffering* logiciel. C+FOX assure aussi la compatibilité de cette zone mémoire avec des outils comme OpenGL, QT ou ALTIVec en garantissant l'alignement mémoire des pixels de l'image et leurs formats de stockage. Cette prise en charge permet alors d'utiliser directement ces données vidéos au sein d'applications complexes.

Listing 2.1 – Acquisition simple via C+FOX

```

1 #include <cfox/cfox.h>
2 using namespace cfox;
3
4 int main()
5 {
6     unsigned char* img;
7     Camera myCam(Mode_640x480_Mono8, FPS_30, 0);
8
9     myCam >> img;
10    processFrame( img );
11
12    return 0;
13 }
```

2.3.2 Gestion de la multi-diffusion

La multi-diffusion FireWire est la fonctionnalité qui permet à BABYLON de fonctionner de manière optimale. Afin de permettre à un ensemble de nœuds de calcul d'accéder aux données vidéos transitant sur le bus FireWire, une de ces machines³ va prendre en charge l'initialisation des communications sur le bus. Les autres machines⁴ vont quant à elles se mettre en attente d'un signal de diffusion. Sur l'ensemble des machines clientes, une instance de Camera est créée. Ceci fait, une synchronisation externe est requise. La machine «serveur» crée alors sa propre instance de Camera. A partir de cet instant, l'ensemble des machines reçoit de manière synchrone les images en provenance de la caméra FireWire.

³que nous désignerons sous le terme de machine «serveur»

⁴que nous désignerons sous le terme de machines «clientes»

En pratique, ces opérations sont réalisées lors de la construction d'une instance de Camera utilisant l'option Synchronized comme illustré sur le listing 2.2 qui s'exécute en mode SPMD — donc sur l'ensemble des nœuds. La construction de la camera est effectuée à la ligne 10, l'option Synchronized permettant de spécifier les éléments spécifiques à la bibliothèque de communication effectuant les tâches de synchronisation.

Listing 2.2 – Multidiffusion via C+FOX

```

1 #include <mpi.h>
2 #include <cfox/cfox.h>
3 using namespace cfox;
4
5 int main(int, char**)
6 {
7     MPI_Init(argc, argv)
8
9     unsigned char* img;
10    Camera myCam(Mode_640x480_Mono8, FPS_30, 0,
11                  Synchronized(MPI_COMM_WORLD, 0));
12
13    myCam >> img;
14    processFrame( img );
15
16    MPI_Finalize();
17    return 0;
18 }
```

Une fois créée, chaque instance de Camera est capable de recevoir de manière synchronisée les données en provenance de la caméra FireWire. C+FOX ne garantit néanmoins que la synchronisation «interne» du flux vidéo — c'est-à-dire la cohérence temporelle du flux à la sortie du capteur — et laisse à l'utilisateur le soin de synchroniser de manière externe son acquisition — c'est-à-dire de s'assurer que les divers nœuds de la machine parallèle effectuent leur acquisition de manière simultanée — en utilisant les primitives de synchronisation à sa disposition, comme MPI_Barrier par exemple.

2.4 Outil de développement MIMD

Plusieurs bibliothèques pour le développement d'applications MIMD sont disponibles. Parmi celles ci, les deux principales sont : PVM⁵ [85] et MPI⁶ [79]. Ces deux technologies reposent sur un modèle commun du passage de message. Le choix entre MPI et PVM est globalement dicté par les meilleures performances globales de MPI, PVM étant plus adapté à l'exécution de programmes parallèles sur des machines hétérogènes.

Même si des projets tentent de concilier les avantages des deux approches en fournissant une bibliothèque alliant les performances de MPI et le support des machines hétérogènes de PVM [65, 64], MPI reste néanmoins la bibliothèque de passage de message la plus utilisée. Elaborée par un consortium d'universitaires et de constructeurs de machines parallèles, MPI représente un consensus sur un ensemble de fonctionnalités nécessaires au développement d'applications parallèles portables. Parmi ces fonctionnalités, on trouve :

- la gestion de communications point à point et collective;
- la portabilité et support de différents langages (C, C++, FORTRAN, Java);
- la possibilité de développer des bibliothèques de haut niveau;
- le support de l'hétérogénéité;
- le support de divers groupes et topologies de processus.

2.4.1 Principes de MPI

De manière générale, un programme MPI consiste en un ensemble de processus autonomes exécutant chacun leur propre code. MPI ne fixe en rien le modèle d'exécution du programme, ni l'interaction avec le système même si, en pratique, la quasi-totalité des implantations parallèles utilisent un modèle SPMD. Dans ce modèle, le même code est dupliqué et s'exécute sur l'ensemble des nœuds de la machine parallèle⁷. Chacune de ces instances est appelé «processus» et peut librement communiquer avec d'autres processus via différentes primitives. Ces primitives constituent les opérations de base de MPI et peuvent être classées en quatre grands groupes.

⁵Parallel Virtual Machine

⁶Message Passing Interface

⁷Nous utiliserons implicitement ce modèle dans le reste de ce manuscrit.

- **Communications point à point.**

MPI fournit un ensemble de fonctions pour l'envoi et la réception de messages constitués de données typées associées à une étiquette optionnelle qui permet de filtrer les messages à la réception. MPI fournit ainsi un large panel de primitives de communication pour l'envoi et la réception synchrone, asynchrone, bloquante et non-bloquante.

- **Communications globales**

Les communications globales effectuent la transmission de données au sein d'un groupe de processus. MPI fournit les primitives suivantes :

- synchronisation d'un groupe de processeurs (`MPI_Barrier`);
- envoi de données vers l'ensemble des processeurs (`MPI_Broadcast`);
- concentration des données vers un processeur unique (`MPI_Gather`);
- dispersion des données vers un groupe de processeur (`MPI_Scatter`);
- réduction des éléments provenant de plusieurs processeurs (`MPI_Reduce`).

- **Support des structures de données utilisateurs**

Toutes les primitives MPI utilisent un argument qui spécifie le type de données transmises. Dans la plupart des cas, cet argument définit un type atomique (`int`, `float`, `double`, ...). Il est néanmoins possible de définir de nouvelles valeurs pour cet argument, valeurs correspondant à une structure utilisateur, en composant les valeurs atomiques. Plusieurs primitives MPI sont disponibles pour permettre l'enregistrement de types composés de zones mémoires contigües ou non ainsi que des structures hétérogènes.

- **Spécification de topologie**

Un des points forts de MPI est de permettre la création de bibliothèques annexes bénéficiant de schémas de communication dédiés. Pour cela, MPI met à disposition un ensemble de primitives permettant d'isoler des sous-groupes de processus au sein de l'ensemble des processus initiaux désigné par la constante `MPI_COMM_WORLD`. Ces primitives se basent sur un objet appelé **communicateur** qui permet d'effectuer ce partitionnement. Toutes les primitives de communication peuvent alors utiliser ces objets pour spécifier dans quel sous-ensemble de processeurs les communications ont effectivement lieu.

2.4.2 Exemple d'utilisation

Le listing 2.3 présente un exemple d'utilisation de MPI dans lequel deux processus communiquent en se transmettant leur rang — c'est-à-dire leur identifiant numérique au sein du groupe de processus auquel ils appartiennent.

Listing 2.3 – Ping-pong MPI

```

1  int main( int argc, char *argv[] )
2  {
3      int rank, size;
4      MPI_Status st;
5      MPI_Init(&argc, &argv);
6      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7      if(rank == 0)
8      {
9          MPI_Send(&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
10         MPI_Recv(&rank, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &st);
11     }
12     else if( rank == 1 )
13     {
14         MPI_Recv(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &st);
15         MPI_Send(&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
16     }
17     MPI_Finalize();
18     return 0;
19 }
```

L'application débute par l'activation du noyau MPI via l'appel à `MPI_Init`. Puis, chaque processus récupère son rang via `MPI_Comm_rank`. Vient ensuite une construction classique dans laquelle le rang du processus permet d'exécuter de manière sélective une portion de code. C'est cette technique qui permet d'émuler un mode d'exécution MIMD au sein d'un modèle SPMD. Ici le processus de rang 0 utilise les primitives d'envoi et de réception `MPI_Send` et `MPI_Recv` pour transmettre une donnée au processus de rang 1. Dans la branche alternative du `if`, le code du processus 1 est exécuté. Ce dernier reçoit la valeur envoyée par le processus de rang 0 et lui transmet la sienne. Enfin, le programme MPI se termine par l'appel à la primitive `MPI_Finalize`

2.4.3 Conclusion

MPI est un outil qui possède de nombreux avantages. Il est portable, permet la création de bibliothèques dédiées et dispose de nombreux outils annexes de débogage, de trace et d'analyse des performances. Néanmoins, le niveau d'abstraction de l'API de MPI reste faible. Ce manque d'abstraction rend le passage d'une application séquentielle à sa version MPI relativement complexe. Dans de nombreux cas, le fait d'avoir à expliciter la topologie des communications rend le code difficile à lire et à maintenir. En outre, des problèmes comme les inter-blocages ou des pertes de performance dus à de mauvais schémas de communications sont monnaie courante et ralentissent d'autant le développement d'applications efficaces. Il existe donc un vrai besoin de fournir une interface de plus haut niveau pour pallier à ces manques.

2.5 Outils de développement pour architectures SMP

Plusieurs outils de développement pour des architectures SMP sont disponibles. Parmi ceux ci, on trouve principalement : MPI, Open MP et pThread.

2.5.1 MPI

L'utilisation de MPI pour développer du code SMP est extrêmement simple. En effet, aucune primitive spéciale ou réécriture de code n'est nécessaire. Pour exécuter un programme SMP avec MPI, il suffit de spécifier une option lors de l'exécution de l'application via `mpirun`⁸. Si l'on considère un programme MPI, son exécution en mode MIMD sur un ensemble de 2 machines mono-processeur est effectuée par l'appel suivant :

```
mpirun -n0-1 mpi_app
```

Pour exécuter cette même application sur une machine comprenant deux processeurs, l'appel de `mpirun` devient :

```
mpirun -c0-1 mpi_app
```

⁸Les applications MPI ont un nom dépendant de l'implantation. Ici, nous utiliseront les notations de LAM-MPI [32] sachant que les techniques présentées sont valides pour d'autre implantations (comme MPICH par exemple).

L'avantage de cette méthode est de permettre de développer des applications SMP de manière simple, en réutilisant les constructions classiques de MPI.

2.5.2 Open MP

Open MP est un standard définissant un ensemble de directives et de fonctions permettant de gérer le partage du calcul entre plusieurs processeurs [42] à travers une interface de haut-niveau. Le modèle d'exécution de Open MP est similaire à celui utilisé par MPI. Des directives et des fonctions insérées dans le programme permettent de gérer la distribution des calculs entre les processeurs en indiquant au compilateur comment paralléliser les sections de code ainsi définies. Le listing 2.4 présente un exemple simple de code OpenMP dans lequel la somme de deux tableaux est effectuée en parallèle.

Listing 2.4 – Exemple de directives Open MP

```

1 #include <omp.h>
2
3 int main()
4 {
5     int i;
6     float a[1000], b[1000], r[1000];
7
8     #pragma omp parallel shared(a,b,r) private(i)
9     {
10         #pragma omp for schedule(dynamic)
11         for (i=0; i < 1000; i++) r[i] = b[i]+a[i];
12     }
13 }
```

Ce code est décomposé en deux parties :

1. Une section dite «parallèle» définie par la directive `omp parallel`. Au sein de cette section, on définit la portée des diverses variables. Ainsi, les tableaux `a`, `b` et `c` sont partagées entre les processeurs – via la directive `shared` – alors que la variable `i` est locale à chaque processeur.
2. Une section de «parallélisation» effective, introduite par la directive `omp for`. Cette section marque le code à suivre comme étant une boucle parallélisable. L'option `schedule(dynamic)` permet de spécifier comment chaque

partie de la boucle sera distribuée sur les différents processeurs. ici, les indices de départ et d'arrivée de la boucle `for` sont modifiés afin de parcourir des sections disjointes des tableaux initiaux.

L'utilisation de Open MP présente un certains nombre d'avantages dans le cadre de la programmation d'architectures SMP. Tout d'abord, son modèle de programmation n'est pas basé sur le passage de message et est donc plus simple : la distribution des données et leur décomposition sont gérées automatiquement via les directives Open MP. Un autre avantage est que le portage d'une application séquentielle vers Open MP peut se faire graduellement en portant section par section le code original tout en fournissant au final des performances satisfaisantes [39]. Les principaux arguments contre l'utilisation de Open MP sur BABYLON résident dans le fait que Open MP nécessite l'utilisation d'un compilateur spécifique. Actuellement, peu de compilateurs – Visual C++ 2005 ou IBM XLC par exemple – supportent Open MP de manière native. En outre, le parallélisme apporté par Open MP est relativement faible car hormis les constructions à base de boucle, peu de constructions sont parallélisable. Son usage est donc limité aux problèmes de parallélisme de données régulier.

2.5.3 pThread

La bibliothèque pThread correspond à une implantation de la norme POSIX 1003.1c. Elle fournit des primitives pour créer des processus légers et les synchroniser. Son interface est divisée en trois grands types de fonctions qui permettent respectivement la création, la destruction et la synchronisation de processus légers, la manipulation de verrous et la manipulation de variables de conditions. Ces fonctions manipulent des concepts bas niveaux et laissent la responsabilité du cycle de vie des processus à l'utilisateur. Le listing 2.5 présente un exemple dans lequel une tâche semblable à celle du listing 2.4 est exécutée.

Dans cet exemple, quatre structures `pthread_t` sont instanciées afin de créer quatre processus via l'appel à `pthread_create`. Cette fonction utilise un pointeur vers un `pthread_t`, un descripteur d'attribut – ici vide –, un pointeur vers la fonction que le processus devra exécuter et un pointeur vers les arguments de cette fonction. Lorsque `pthread_create` termine, le *thread* correspondant est activé. Pour permettre la synchronisation finale de l'ensemble des processus, nous utilisons la fonction `pthread_join`. Contrairement à Open MP, pThread n'impose pas de modèle de programmation spécifique. Le spectre d'applications parallélisable est donc beaucoup plus large en permettant d'exprimer des problèmes de

parallélisme de données ou de tâches. Néanmoins, l'interface de pThread reste de très bas niveau, nécessitant l'utilisation de primitives C peu sûres.

Listing 2.5 – Exemple d'utilisation de pThread

```

1 #include <pthread.h>
2 struct arg { float *a,*b,*r; };
3
4 void* func(void* in)
5 {
6     arg* p = (arg*)(in);
7     for(int i=0;i<250;i++)
8         p->r[i] = p->a[i]+p->b[i];
9     return NULL;
10 }
11
12 int main()
13 {
14     float a[1000],b[1000],r[1000];
15     pthread_t t[4];
16     th_arg arg[4];
17
18     for(int i=0;i<4;i++)
19     {
20         arg[i].pa = &a[i*250];
21         arg[i].pb = &b[i*250];
22         arg[i].pr = &r[i*250];
23     }
24
25     for(int i=0;i<4;i++)
26         pthread_create(&t[i],NULL,func,&arg[i]);
27
28     for(int i=0;i<4;i++)
29         pthread_join(t[i],NULL);
30 }
```

Une implantation orientée objet est proposée au sein de la bibliothèque C++ BOOST [89]. Cette interface de plus haut niveau encapsule la notion de processus léger, de mutex et de conditions dans un ensemble de classes dont la sémantique évite les erreurs classiques de la programmation concurrente [17]. L'exemple précédent s'exprime alors comme présenté sur le listing 2.6.

Listing 2.6 – Exemple d'utilisation de *BOOST::Thread*

```

1 #include <boost/thread/thread.hpp>
2
3 class Func
4 {
5     public:
6         Func( float* pa, float* pb, float* pr, int i )
7             : a(pa), b(pb), r(pr), idx(i) {}
8
9     void operator()()
10    {
11        for(int i=250*idx;i<250*(idx+1);i++)
12            r[i] = a[i]+b[i];
13    }
14
15     protected:
16     float *a,*b,*r;
17     int idx;
18 };
19
20 int main(int argc, char* argv[])
21 {
22     float a[1000],b[1000],r[1000];
23     boost::thread_group t;
24
25     for(int i=0;i<4;i++)
26         t.create_thread(new Func(a,b,r,i));
27
28     t.join_all();
29 }
```

Dans cette écriture, la fonction effectivement exécutée par le processus léger est passée via une classe foncteur. La classe `thread_group` permet de créer et synchroniser plusieurs *threads* et s'occupe de gérer l'ensemble des appels interne à pThread. La question se pose alors de savoir si l'utilisation de pThread en lieu et place de MPI pose des problèmes de performances. On montre [57] que la différence de temps d'exécution entre une application MIMD-SMP utilisant seulement MPI et une application MIMD-SMP utilisant conjointement MPI et pThread était négligeable pour des problèmes de grandes tailles.

2.5.4 Conclusion

Le choix d'un outil de développement SMP pour BABYLON est relativement complexe. Plusieurs facteurs sont à prendre en compte :

- Open MP fournit des performances très satisfaisantes sur des *clusters* de machines SMP [116] tout en étant plus simple d'accès que pThread.
- L'homogénéité fournie par MPI dans l'écriture d'applications MIMD-SMP est un atout non négligeable dans le cycle de développement d'applications parallèles .
- Les fonctionnalités bas niveau de pThread permettent de paralléliser des schémas complexes que les problèmes réguliers supportés par OpenMP.

Globalement, l'intérêt de coupler MPI et Open MP pour permettre le développement d'application sur une machine hybride MIMD-SMP ne devient flagrant que lorsque les temps de communications sont prépondérants – ce qui n'est plus le cas dans le cadre de notre *cluster* pourvu d'un double réseau. En outre, dans la plupart des cas [35], l'approche utilisant seulement MPI fournit de meilleur performance. Elle permet aussi de limiter le nombre de modèle d'exécution à maîtriser au sein d'une code source. Par contre, elle nécessite un découpage relativement complexe de l'application parallélisée et est incompatible avec les primitives bas-niveaux utilisées par C+FOX .

L'ensemble de ces considérations nous amènent alors à utiliser MPI et pThread. Tout comme l'approche combinant MPI et OpenMP, cette solution permet de simplifier l'écriture du code parallèle en séparant fortement les parties MIMD des parties SMP. Elle est néanmoins plus flexible, plus simple d'emploi grâce à l'interface orientée objet fournie par BOOST et permet d'intégrer l'utilisation de C+FOX au sein d'une application MIMD-SMP.

2.6 Outil de développement SIMD

Comme présenté dans la section 2.2.3, l'utilisation du noyau SIMD du PPC 970 passe par l'extension ALTIVEC. Son intégration est facilitée par l'existence de compilateurs C standards gérant l'utilisation des registres vectoriels ALTIVEC. Ce type de compilateur est donc un outil de développement facile à appréhender, l'écriture d'une fonction C utilisant ALTIVEC se faisant sans recours à l'assembleur. En pratique, 162 primitives C sont disponibles et se repartissent en six grands groupes [58].

- **Les instructions de chargement.**

Elles permettent d'accéder en lecture ou en écriture au contenu d'un vecteur ou d'un élément d'un vecteur, alignés ou non.

- **Les instructions de «*prefetch*».**

Ces fonctions autorisent la manipulation directe des informations présentes dans la mémoire cache de manière à optimiser leur ordre de traitement et de ce fait augmenter les performances globales de l'application.

- **Les instructions de manipulation des vecteurs.**

Certainement les fonctions les plus puissantes d'ALTIVEC, on trouve dans cette catégorie toutes les instructions de décalages de bits, de permutations d'éléments, de fusion, de duplication de données vectorielles.

- **Les instructions arithmétiques.**

Classiquement ces fonctions permettent d'effectuer des opérations telles que l'addition, la soustraction ou des opérations sur les nombres en virgule flottante (arrondis, troncature) sur des vecteurs. ALTIVEC fournit aussi un certain nombre d'instructions émulant des fonctions de type DSP.

- **Les instructions logiques.**

Pendant des opérations arithmétiques, on trouve ici une batterie de fonctions basées sur les opérations booléennes bits à bits.

- **Les instructions de comparaison.**

Ces fonctions permettent de paralléliser des traitements équivalents à diverses constructions de type if ... then ... else

L'utilisation d'ALTIVEC au sein de code C est alors relativement simple. Considérons par exemple une fonction C calculant la différence binaire de deux images de $N \times N$ pixels (listing 2.7). Il s'agit ici d'effectuer la soustraction des intensités des pixels et d'y appliquer un seuillage pour obtenir une image en noir et blanc.

Listing 2.7 – Différence d'images en C

```

1 unsigned char I1[N*N], I2[N*N], R[N*N];
2 for( size_t i = 0; i < N*N; i++ )
3 {
4     signed short s = I2[i] - I1[i];
5     R[i] = (s<10 || s>240) ? 0 : 255;
6 }
```

Examinons le code équivalent (listing 2.8) écrit en utilisant les primitives C fournies par ALTIVEC. La version vectorisée reprend telle quelle la structure de l'algorithme C mais exhibe un certain nombre de différences :

- La nécessité de générer les constantes vectorielles en amont de la boucle principale. La création d'une constante via la primitive `vec_splat` est en effet relativement coûteuse.
- Les primitives `vec_ld` et `vec_st` remplacent les accès aux éléments du tableau en effectuant respectivement le chargement ou l'écriture de blocs de 16 éléments.
- Chaque primitive ALTIVEC traite les données par blocs de 16 éléments, ce qui implique que la boucle principale ne doit plus effectuer $N \times N$ itérations, mais seulement $\frac{N \times N}{16}$.
- Les tests nécessaires au seuillage sont aussi vectorisés grâce à la primitive `vec_sel` qui permet d'effectuer l'équivalent de 16 sélections en trois cycles seulement.

Quel est alors l'impact de cette réécriture sur les performances de cette fonction ? Des mesures effectuées sur cette implantation montre que la version vectorisée de cette fonction subit une accélération de l'ordre de 12 par rapport à la fonction C originale.

Listing 2.8 – Différence d’images en C ALTIVEC

```

1 unsigned char I1[N*N], I2[N*N], R[N*N];
2 vector unsigned char vS,vR,v240,v10,v0,v255;
3 vector bool char vC;
4
5 v0 = vec_splat_u8(0);
6 v10 = vec_splat_u8(10);
7 v240 = (vector unsigned char)vec_splat_s8(-15);
8 v255 = (vector unsigned char)vec_splat_s8(-1);
9
10 for( size_t i = 0; i < (N*N)/16; i++ )
11 {
12     vS = vec_sub( vec_ld(I2,16*i),vec_ld(I1,16*i));
13     vC = vec_or(vec_cmplt( vS, v10 ),vec_cmpgt( vS, v240 ))
14         ;
15     vR = vec_sel(v255,v0,vC);
16     vec_st( vR,R,i*16);
17 }
```

Néanmoins, la modification de la taille de l’image induit des variations non négligeable de ce gain, ce qui pose la question de l’évaluation globale des performances de l’extension ALTIVEC.

2.6.1 Performances

L’extension ALTIVEC a fait l’objet de différents travaux afin de déterminer ses performances lors de sa mise en oeuvre sur plusieurs types d’algorithmes. Parmi ces travaux, les recherches menés par Julien Sébot [155] et Lionel Damez [52] ont permis de quantifier l’apport en termes de performance d’ALTIVEC au sein d’applications réalistes. Les tests effectués sont :

- un algorithme de conversion RGB/YCbCr traitant des images 2048x1536;
- un algorithme de filtrage numérique appliqué sur un signal 1D de 10Mo;
- un algorithme de filtrage numérique appliqué sur un signal 2D de 16Mo;
- un algorithme de calcul de produit matriciel (DAXPY);
- un algorithme de recherche du maximum d’un tableau et de l’indice associé;
- un algorithme de rendu 3D utilisant un modèle d’illumination de Phong.

Pour chaque test, le tableau 2.3 présente le gain obtenu par rapport à une implantation en C, la valeur maximale théorique du gain qu'ALTIVEC est en mesure de fournir pour cette application⁹ et l'efficacité de l'implantation utilisant ALTIVEC — c'est-à-dire, le rapport entre le gain effectivement mesuré et le gain théorique maximal.

Algorithme	RGB2YCbCr	FIR 1D	FIR 2D	DAXPY	MAX	3DKERNEL
Gain	5.98	3.2	10.99	2.98	4.98	2.95
Gain Théorique	8	4	16	4	4	4
Efficacité	74.75%	80%	68.69%	74.5%	124.5%	73.75%

TAB. 2.3 – Evaluation des performances d'ALTIVEC [155, 52]

Plusieurs phénomènes sont à noter. Tout d'abord, le gain fourni par ALTIVEC est dans tous les cas supérieur à 70% du gain maximal théorique. Cette efficacité est d'autant plus grande que le type d'élément contenu dans le vecteur est petit. Il est donc ainsi plus facile de profiter pleinement du gain théorique de 16 offert par les vecteurs d'entiers 8 bits que de profiter du gain de 4 des entiers 32 bits. Enfin, on note des gains supérieurs au maximum théorique, principalement avec les nombres réels.

2.6.2 Mise en oeuvre

L'analyse détaillée de ces résultats et des codes sources des applications associées permet de dégager un certains nombre de recommandations pour tirer parti de manière optimale de l'extension ALTIVEC :

- **Eviter les accès à un unique élément d'un vecteur**

Extraire une valeur d'un vecteur ALTIVEC nécessite plusieurs étapes relativement coûteuses : lire la valeur de l'élément, le transférer dans l'unité scalaire et l'écrire en mémoire. Enfin, lorsque les calculs sur cet élément sont terminés, il convient de l'extraire de la mémoire et de le transférer dans un registre vectoriel. L'ensemble de ces opérations nécessite de sortir plusieurs fois des chemins de données vectoriels et scalaires et de passer par la mémoire centrale. Or, l'accès à cette mémoire induit une latence 30 à 50 fois plus élevée que celle induite par un accès au sein des caches L1 ou L2.

⁹En fonction du type de donnée traitée.

- **Utiliser préférentiellement des structures de données uniformes**

Considérons un problème mettant en œuvre une structure de données définies par l'utilisateur comme, par exemple, le problème du retour à un repère affine d'un vecteur 3D exprimé en coordonnées homogènes :

$$(X, Y, Z, W) \longrightarrow \left(\frac{X}{W}, \frac{Y}{W}, \frac{Z}{W} \right)$$

Une manière intuitive de représenter ce type de données en utilisant des vecteurs ALTIVEC est d'utilisé un **tableau de structures** :

`vector float V[] = [X0 | Y0 | Z0 | W0] ... [XN | YN | ZN | WN]`

Pour effectuer ce calcul, il faut alors extraire la valeur de la composante W de chacun de ces vecteurs afin d'effectuer la division de chacune des autre composante de la structure. Or, nous venons de voir qu'une telle extraction a un coût important. Une solution plus efficace consiste non pas à stocker ces données sous forme de **tableau de structures** mais sous forme d'une **structure de tableaux**.

<code>vector float V[4][] =</code>	<table border="1"> <tr><td>X₀</td><td>X₁</td><td>...</td><td>X_N</td></tr> <tr><td>Y₀</td><td>Y₁</td><td>...</td><td>Y_N</td></tr> <tr><td>Z₀</td><td>Z₁</td><td>...</td><td>Z_N</td></tr> <tr><td>W₀</td><td>W₁</td><td>...</td><td>W_N</td></tr> </table>	X ₀	X ₁	...	X _N	Y ₀	Y ₁	...	Y _N	Z ₀	Z ₁	...	Z _N	W ₀	W ₁	...	W _N
X ₀	X ₁	...	X _N														
Y ₀	Y ₁	...	Y _N														
Z ₀	Z ₁	...	Z _N														
W ₀	W ₁	...	W _N														

Ici, il suffit donc de stocker chaque composante du vecteur dans un tableau séparé et de travailler séparément sur chacun de ces tableaux. Dans cette représentation, le calcul peut alors se faire de manière régulière sans avoir à extraire les valeurs de W .

Ce changement de point de vue est une étape critique du processus de vectorisation car une mauvaise mise en correspondance entre les structures de données du problème initial et leur représentations vectorielles réduit considérablement les performances du code ALTIVEC.

- **Tirer partie de l'architecture matérielle de l'extension ALTIVEC**

Si l'on désire tirer parti des avantages architecturaux d'ALTIVEC, il convient de fournir à ses différentes unités de calcul et à leurs pipelines respectifs une quantité de données suffisante pour les occuper de manière ininterrompue.

Reprendons par exemple le code du listing 2.8. Si l'on ignore la partie gérant la boucle `for` et les primitives de chargements, le code assembleur correspondant se résume à l'appel des primitives ALTIVEC `vsububm`, `vcmpgtub`, `vcmpltub`, `vor` et `vsel`. Ces routines traversent un pipeline de 3 étages et comme leurs appels successifs dépendent les uns des autres, il doit s'écouler 15 cycles complets avant que le résultat suivant puisse être calculé. L'état du pipeline de l'unité de calcul sur les entiers d'ALTIVEC peut alors être représenté comme sur le tableau 2.4.

Cycle	Etage 1	Etage 2	Etage 3
1	<code>vsububm</code>		
2		<code>vsububm</code>	
3			<code>vsububm</code>
4	<code>vcmpgtub</code>		
5		<code>vcmpgtub</code>	
6			<code>vcmpgtub</code>
7	<code>vcmpltub</code>		
8		<code>vcmpltub</code>	
9			<code>vcmpltub</code>
10	<code>vor</code>		
11		<code>vor</code>	
12			<code>vor</code>
13	<code>vsel</code>		
14		<code>vsel</code>	
15			<code>vsel</code>

TAB. 2.4 – Etat du pipeline ALTIVEC sans optimisation

Sous cette forme, ce code vectorisé ne tire pas partie de la totalité de la puissance fournie par l'extension ALTIVEC . De nombreux cycles sont «gachés» à attendre la fin de l'exécution des instructions chargées dans le pipeline. Si l'on réécrit ce code de manière à effectuer non pas un mais trois calculs par itération de la boucle, le pipeline ALTIVEC prend alors la forme représentée sur le tableau 2.5. Dans la version originale, nous calculons la différence de deux vecteurs en 15 cycles. Dans la version déroulée, nous calculons trois différences de deux vecteurs en 17 cycles, soit un gain de l'ordre de 2,6. Ce facteur se combine au gain dû uniquement à la nature SIMD d'ALTIVEC et nous permet soit de masquer les faibles performances dues à la taille des données traitées, soit de bénéficier d'un gain supérieur au gain maximum théorique. Il arrive néanmoins que la vectorisation «naïve» d'un algorithme

remplisse de manière optimale les pipelines ALTIVEC . On observe alors une phénomène semblable à celui exposé dans la section 2.6.1.

Cycle	Etage 1	Etage 2	Etage 3
1	vsububm		
2	vsububm	vsububm	
3	vsububm	vsububm	vsububm
4	vcmpgtub	vsububm	vsububm
5	vcmpgtub	vcmpgtub	vsububm
6	vcmpgtub	vcmpgtub	vcmpgtub
7	vcmpltub	vcmpgtub	vcmpgtub
8	vcmpltub	vcmpgtub	vcmpgtub
9	vcmpltub	vcmpltub	vcmpltub
10	vor	vcmpltub	vcmpltub
11	vor	vor	vcmpltub
12	vor	vor	vor
13	vsel	vor	vor
14	vsel	vsel	vor
15	vsel	vsel	vsel
16		vsel	vsel
17			vsel

TAB. 2.5 – Effet d'un déroulage partiel sur le pipeline ALTIVEC

La stratégie optimale est donc d'écrire du code vectoriel qui maximise l'utilisation des différents pipelines fournis par ALTIVEC . En général [142, 52], il convient de dérouler ces algorithmes d'un facteur multiple de la taille du pipeline. Ces tailles dépendent de l'unité de calcul et sont données dans le tableau 2.6.

Unité	Taille du Pipeline
Unité de permutation (VPERM)	2
Unité entière (VSIU)	3
Unité entière composite (VCIU)	5
Unité flottante (VFPU)	8
Unité de chargement (LSU)	4

TAB. 2.6 – Taille des pipelines de l'unité ALTIVEC

Ainsi, un code utilisant de manière intensive l'unité flottante gagnera à être déroulé d'un facteur 8 par exemple. Dans des cas plus complexes, la valeur

du pas de déroulage devient plus difficile à déterminer du fait du caractère super-scalaire de l'unité ALTIVEC et de la grande disparité des tailles de ces pipelines.

Au final, l'ensemble de ces recommandations a permis de définir une forme optimale des algorithmes vectoriels qui permet de maximiser le gain fourni par ALTIVEC. Ollman [142] propose ainsi un modèle dit du «blitter», qui consiste à parcourir de manière régulière les zones de données d'entrées, y effectuer une quantité de calcul importante, dérouler ces opérations d'un pas compatible avec la profondeur du ou des pipelines utilisés et finalement de stocker le ou les résultats dans une ou plusieurs zones mémoire de destination. Malheureusement, de nombreux algorithmes séquentiels – comme l'étiquetage en composantes connexes ou la transposition de matrice rectangulaire – ne sont pas transposables dans ce modèle [52, 142] de manière triviale. Il est alors nécessaire d'optimiser de manière spécifique chacune de ces fonctions afin de conserver des performances élevées.

2.6.3 Conclusion

L'extension ALTIVEC apporte un gain d'efficacité non négligeable aux architectures à base de PPC 970. Néanmoins, son utilisation comporte un certain nombre de contraintes.

- **La nécessité de garder à l'esprit l'architecture de l'extension.**

Si l'utilisation simple d'ALTIVEC peut conduire à des gains satisfaisant, il est nécessaire de tenir compte des problématique de remplissage des pipelines internes et de la nature super-scalaire de l'extension en elle-même pour obtenir des gains proches ou supérieurs au maximum théorique.

- **L'absence de support pour les réels double précision.**

ALTIVEC ne permet de manipuler que des entiers et des nombres en virgules flottantes simple précision. Cette absence force de nombreux développeurs à reformuler leurs algorithmes pour conserver leur précision ou leur robustesse [151, 52] ou d'utiliser des types de données *ad hoc* [120].

- **Un support restreint des conversions entre types de données.**

Les conversions entre vecteurs de types différents n'est pas automatique. L'écriture d'un transtypage classique ne fait que modifier le registre dans lequel le vecteur sera stocké mais ne provoque aucune conversion. Pour effectuer correctement ces conversions, un ensemble de primitives spéci-

fiques sont fournies mais leur utilisation complique d'autant l'écriture du code vectorisé.

- **L'asymétrie du jeu de primitives.**

Même si les 162 primitives fournies par ALTIVec permettent de couvrir les besoins classiques du calcul scientifique, un certain nombre d'opérations simples manquent à l'appel. Ainsi, il n'existe pas de primitive de multiplication pour les entiers 32 bits ou 8 bits, de division entière ou de calcul de fonctions trigonométriques. En outre, certaines primitives s'appliquant sur des nombres réels ne fournissent que des résultats approximatifs.

- **Des contraintes sur le stockage des données en mémoire.**

Les données numériques stockées en mémoire doivent suivre deux contraintes pour être chargées efficacement en mémoire : elles doivent être contiguës et alignées sur des adresses multiples de 16 octets. Si une de ces contraintes n'est pas respectée, les performances chutent de près de 40% [3, 52].

Tout cela fait que l'écriture d'un code vectoriel tirant parti de manière optimale du potentiel de l'extension ALTIVec est complexe. La nécessité d'un modèle de programmation de plus haut niveau qui encapsulerait l'ensemble de ces contraintes est donc bien réelle.

2.7 Application : Stabilisation de flux vidéo

Afin de valider l'architecture matérielle et logicielle de BABYLON, nous avons implémenté une application de vision de complexité moyenne et comparé ses performances séquentielles et parallèles. L'application choisie est une application de stabilisation d'images [57]. La stabilisation d'images joue un rôle important dans plusieurs systèmes de vision artificielle, comme la télé-opération, la robotique mobile, la reconstruction de scène, la compression des vidéos, la détection d'objets en mouvement et beaucoup d'autres. Chacune de ces applications a ses spécificités, et le concept de "stable" peut donc varier. Nous nous intéressons au cas général d'une caméra embarquée sur un système mobile, et fixée à celui ci de façon rigide. Cette configuration est fréquemment présente dans des systèmes de télé-opération ou d'aide à la conduite. La séquence d'images issue de cette caméra contient des informations concernant le déplacement du véhicule dans son environnement. Ce déplacement peut être scindé en deux composantes : celle due aux mouvements commandés du véhicule, et celle issue des mouvements parasites

(non-commandés) subis par la caméra (rugosité du terrain, vibrations, etc.). Cette composante parasite est, d'un point de vue signal, de fréquence plus élevée que le mouvement principal de la caméra. La stabilisation d'images va donc consister à appliquer un filtre numérique passe-bas. La composante parasite peut, selon son amplitude, nuire de façon très significative à la visualisation et à la compréhension des images, soit par un observateur humain, soit par un système de vision artificielle. Dans ce cas, la stabilisation de la séquence consiste dans l'élimination ou l'atténuation de la seconde composante du mouvement, tout en conservant la composante issue des mouvements commandés. On parle donc de stabilisation sélective. Cette classe d'application met en œuvre de nombreuses techniques de traitement d'images bas niveau et des méthodes d'estimation du mouvement 2D ou 3D de la caméra.

2.7.1 Description de l'algorithme

Plusieurs méthodes de stabilisation automatique d'image ont été proposées. Ces méthodes peuvent être classées dans trois familles principales, selon le modèle d'estimation du mouvement adopté : les méthodes 2D ou planaires [135], les méthodes 3D [61] et les méthodes 2,5D [104]. Ces algorithmes de stabilisation sont constitués d'une séquence de traitements de différents niveaux, appliqués sur les images successives de la séquence vidéo. Généralement, trois modules de traitement sont présents :

- 1. la mise en correspondance entre les images :**

Pour la mise en correspondance des images, on détecte et suit un ensemble de primitives visuelles. La technique consiste, dans un premier temps, à localiser dans l'image i des régions riches en information visuelle et dans un deuxième temps, à retrouver ces mêmes régions à l'image $i+1$. Après avoir détecté ces primitives, il faut être capable de les retrouver dans une autre image. Cela est fait à l'aide d'une fonction de corrélation et d'une stratégie de recherche multi-résolution.

- 2. l'estimation du déplacement global :**

Une fois que les images successives de la séquence ont été mises en correspondance, il s'agit de déterminer les paramètres décrivant le mouvement entre les images selon le modèle de mouvement choisi.

- 3. la correction/compensation des mouvements :**

Finalement, après l'estimation du mouvement global entre les images, il faut procéder à la correction de sa composante non voulue. Cette dernière étape du traitement est fortement liée à l'application, car c'est elle qui définira quelle partie du mouvement doit être conservée [104].

2.7.2 Implantation séquentielle

Nous avons développé [57] une méthode de stabilisation basée sur un modèle de mouvement 2D, avec extraction de primitives par ondelettes de Harr. Comme dans [192], le calcul des ondelettes est effectuée à partir d'une image intégrale afin de réduire le temps de calcul. Deux images de résolution inférieure ($1/4$ et $1/2$) sont aussi obtenues à partir de l'image intégrale, permettant la recherche des points correspondants avec une approche pyramidale sur trois niveaux de résolution. Celle-ci est ensuite effectuée en appliquant des fonctions SSD¹⁰ entre la primitive recherchée d'une image et ses correspondants potentiels dans l'image suivante. Une fois obtenus les appariements de points entre deux images successives, les paramètres du modèle 2D sont estimés par une méthode robuste aux faux appariements : une approximation par Moindres Carrés Médians. Ces paramètres τ_x , τ_y et θ représentent les translations en x et en y et la rotation autour de l'axe z de l'image d'entrée. Finalement en possession des paramètres du déplacement, ceux-ci sont traités par un filtre numérique passe-bas, afin d'isoler la composante parasite du mouvement, qui est ensuite appliquée comme correction à l'image correspondante dans le but de stabiliser la séquence (fig. 2.8).

2.7.2.1 Construction de l'image intégrale

A partir de chaque image en niveaux de gris reçue depuis la caméra, on calcule une image intégrale, qui sera utilisée pour la détection par ondelettes et pour la génération des images sous-échantillonées. L'image intégrale contient, à la position (X, Y) , la somme de tous les pixels contenus à l'intérieur du rectangle borné par les pixels de coordonnée $(0, 0)$ et (X, Y) dans l'image originale. À partir de l'image intégrale, la somme de tous les pixels dans une zone rectangulaire peut être obtenue en quatre accès à l'image intégrale et 3 additions, indépendamment du nombre de pixels [192]. Cette propriété permet lors de la phase de détection par ondelettes et le calcul d'images sous échantillonées de minimiser les calculs de sommes effectués sur l'image.

¹⁰Sum of Squared Differences

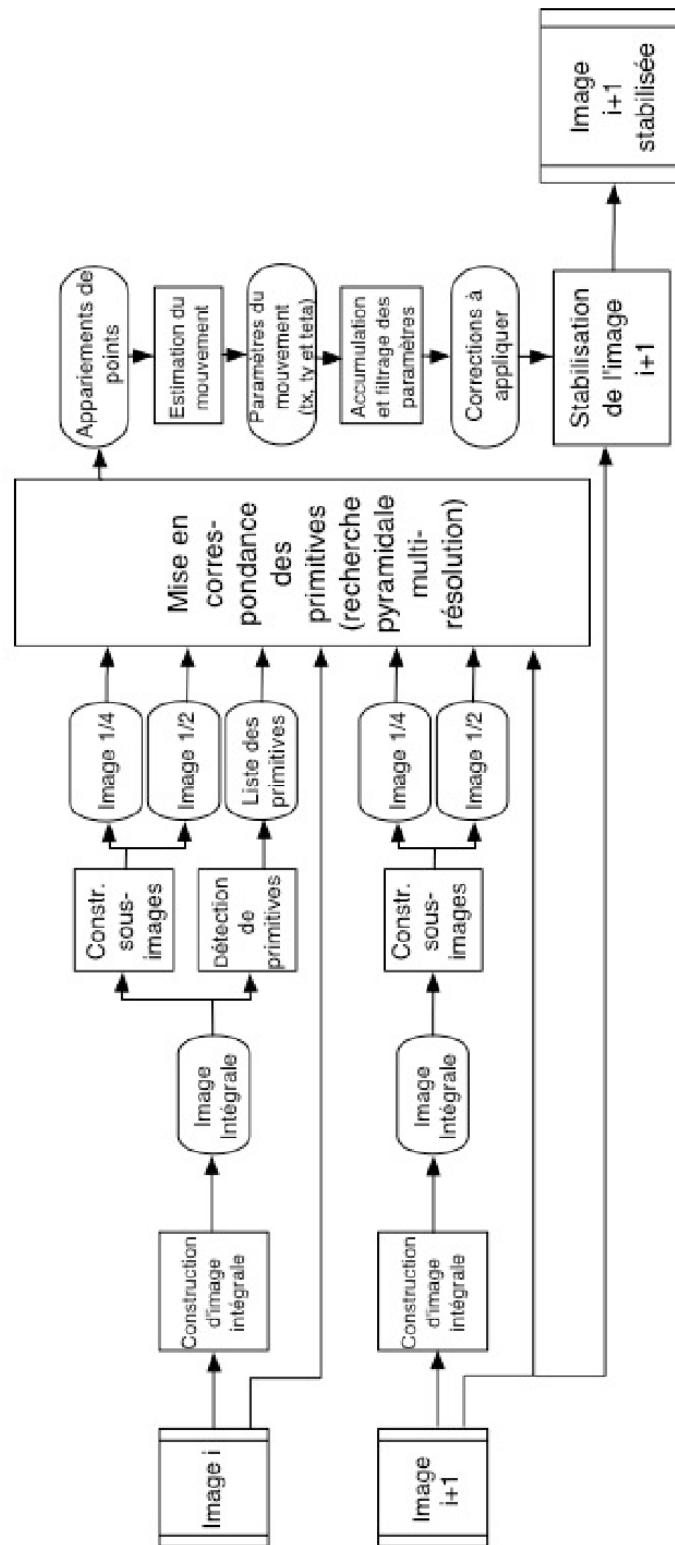


FIG. 2.8 – Synopsis de l'algorithme de stabilisation

2.7.2.2 Extraction des primitives

Les primitives sont recherchées en appliquant les ondelettes dans une zone de détection de taille $q \times r$. Nous utilisons la moitié supérieure de l'image, ce qui dans une scène d'extérieur permet de détecter les primitives situées à l'horizon. Ces régions sont normalement très éloignées de la caméra, ce qui permet de respecter au mieux la contrainte de scène planaire imposée par le modèle 2D. La zone de recherche est divisée en $\frac{n}{3}$ bandes verticales de taille $\frac{3q}{n} \times r$, où n est le nombre de primitives recherchées. Par exemple, avec une image de format 1280×960 , on pose $q = 1024$, $r = 384$ ce qui nous donne des blocs de 128×384 pour $n = 24$ (fig. 2.9). Chaque bande est balayée par une ondelette verticale, une ondelette horizontale et une ondelette diagonale. Pour chaque type d'ondelette, le point qui retourne la valeur de gradient la plus élevée est retenu comme primitive pour le suivi.

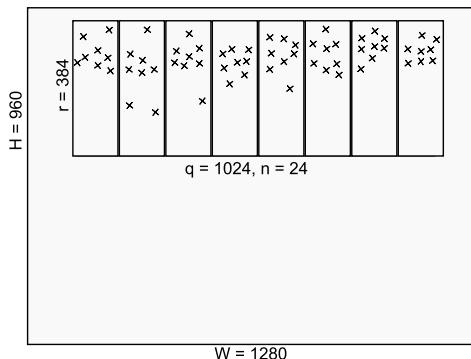


FIG. 2.9 – Découpage de l'image pour la détection des points d'intérêts

2.7.2.3 Mise en Correspondance de primitives

Considérant que l'étape de détection a été réalisée sur l'image i , nous allons maintenant rechercher dans l'image $i + 1$ les correspondants des n primitives retenues. Une fenêtre de recherche de taille $2T \times 2T$ est définie autour de la position où la primitive a été détectée (fig. 2.10). Ensuite, la somme des distances (SSD) entre chaque région à l'intérieur de cette fenêtre et le motif retenu comme primitive dans l'image i est calculée. La région de l'image $i + 1$ qui minimise le SSD est considérée comme étant la correspondante à cette primitive. Nous avons donc un appariement de points entre les deux images consécutives. Cette opération est réalisée pour chacune des n primitives détectées.



FIG. 2.10 – Mise en correspondance des primitives entre deux images à stabiliser

Afin de diminuer le nombre d’opérations à réaliser, la recherche est faite au moyen d’une approche multi résolution. Nous utilisons d’abord une image échantillonnée à l’échelle $\frac{1}{4}$, et la recherche est faite à l’intérieur d’une fenêtre de taille $\frac{T}{2} \times \frac{T}{2}$. Cela nous donne une première estimation de la position du correspondant. À partir de cette estimation, est effectuée une deuxième recherche sur l’image sous-échantillonnée à l’échelle $\frac{1}{2}$. Cette fois, la recherche est faite à l’intérieur d’une fenêtre 3×3 autour de la position estimée précédemment. Une deuxième estimation est donc obtenue, plus précise que la première. Finalement, une dernière recherche est faite sur l’image originale, cette fois avec une précision sous-pixel (de $\frac{1}{8}$ de pixel). Une fenêtre 2×2 est analysée, et la valeur des points entre les pixels est calculée au moyen d’une interpolation bilinéaire avec les valeurs des pixels adjacents.

2.7.2.4 Estimation des paramètres du modèle 2D

En possession des n appariements de points entre les images i et $i + 1$, les paramètres du modèle 2D, décrivant le mouvement d’une image à l’autre, sont estimés. Il est supposé que le déplacement peut être modélisé par une matrice de transformation homogène, constituée d’une rotation autour de l’axe optique (Θ), d’une translation horizontale (τ_x) et d’une translation verticale (τ_y). Le déplacement entre deux images d’un point (x, y) est donc décrit par la relation :

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & \tau_x \\ \sin\theta & \cos\theta & \tau_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Les trois paramètres inconnus de la matrice (θ , τ_x et τ_y) sont estimés en appliquant le modèle aux n appariements de points retrouvés à l'étape de mise en correspondance. Notons \mathcal{P}_1 le nuage de points d'intérêt détectés à l'image i et \mathcal{P}_2 le nuage formé par leurs correspondants trouvés à l'image $i + 1$. Les deux nuages de points sont centrés par rapport à leurs barycentres respectifs, donnant deux nuages centrés p'_1 et p'_2 , liés par une rotation autour de l'origine : $p'_2 = R.p'_1$. Il faut donc minimiser le critère :

$$S = \sum_i \|p'_{2i} - Rp'_{1i}\|^2$$

La minimisation du critère S donne le paramètre θ , et une fois en possession de l'angle de rotation les paramètres de translation sont déduits à partir du mouvement des barycentres des deux nuages. Dans les équations ci-dessous, (bx_1, by_1) et (bx_2, by_2) sont les coordonnées du barycentre des nuages p_1 et p_1 respectivement :

$$\tau_x = bx_2 - bx_1 \cdot \cos\theta - by_1 \cdot \sin\theta$$

$$\tau_y = by_2 + bx_1 \cdot \sin\theta - by_1 \cdot \cos\theta$$

L'estimation des paramètres du mouvement est réalisée au moyen d'un filtrage par les moindres carrés médians.

2.7.2.5 Accumulation et filtrage des paramètres

Les valeurs θ , τ_x et τ_y calculées précédemment sont ensuite combinées à la matrice \mathbf{M}_i , contenant les transformations subies par l'image depuis le début de la séquence pour obtenir la matrice de transformation \mathbf{M}_{i+1} , qui décrit le déplacement de l'image $i + 1$ par rapport au repère de référence.

Un filtre linéaire du premier ordre est appliqué à chaque paramètre indépendamment, les coefficients des trois filtres pouvant être réglés par l'utilisateur. Cela permet d'obtenir une stabilisation plus ou moins forte, selon les contraintes de l'application. Le découplage des filtres permet encore d'avoir différents niveaux de stabilisation pour la rotation et les translations.

Finalement, les valeurs filtrées sont utilisées pour obtenir la matrice de déplacement inverse, qui est appliquée à l'image $i + 1$ afin de la stabiliser, c'est-à-dire de la ramener vers un repère de référence. La figure 2.11 présente le résultat de l'application de la chaîne complète de stabilisation sur une séquence de synthèse.

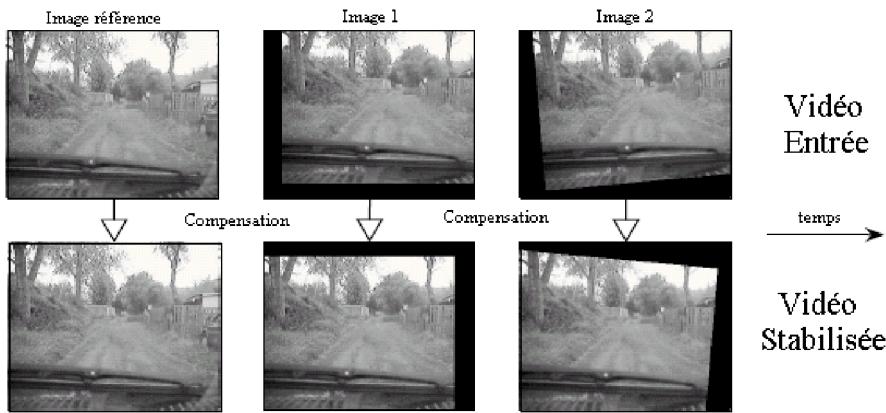


FIG. 2.11 – Application de la stabilisation à un flux vidéo

2.7.2.6 Performances

Les temps d'exécution de cet algorithme sont résumés dans le tableau 2.7. Dans une optique d'utilisation au sein d'un dispositif mobile, on note clairement que excepté pour des images de petites tailles, l'exécution en temps réel est impossible. Pour des images plus grandes, ce temps d'exécution devient rapidement très important et rend impraticable l'application de cet algorithme. Il est donc bien nécessaire de fournir une implantation parallèle.

Taille	320x240	640x480	1280x960	2560x1920	5120x3840
Temps	20.31 ms	82.40 ms	148.4 ms	393.80 ms	1353 ms

TAB. 2.7 – Temps d'exécution de l'algorithme de stabilisation

2.7.3 Stratégie de parallélisation

Afin de réaliser une implantation parallèle efficace, nous avons analysé les temps d'exécution en mode séquentiel de chacune des étapes de l'algorithme (tab. 2.8). Il apparaît que les étapes de détection et de suivi des primitives (étapes 3 et 4) ont le temps d'exécution le plus important. Cependant, avec l'augmentation de la taille de l'image, l'importance relative du calcul des images intermédiaires (1 et 2) augmente. Une parallélisation de ces étapes est donc aussi souhaitable. Nous avons donc choisi de paralléliser ces quatre premières étapes. Les deux dernières étapes, ne présentant pas une complexité importante (moins de 2% du temps d'exécution total), resteront séquentielles.

Etapes	320x240	640x480	1280x960	2560x1920	5120x3840
Étapes 1 et 2	1%	2%	6%	8%	9%
Étapes 3 et 4	97%	96%	93%	91%	90%
Étapes 5 et 6	2%	2%	1%	1%	1%

TAB. 2.8 – Importance relative des étapes de la stabilisation.

Il est ensuite nécessaire de déterminer quelle forme de parallélisme est exploitable dans chacune de ces étapes. L’analyse du code séquentiel (tableau 2.9) montre que les étapes 1 à 4 exposent à la fois un parallélisme de tâches et de données.

Étapes	Parallélisme intrinsèque	Ratio calcul/mémoire	Volume de données
(1) Calcul de l’image intégrale	données	faible	élevé
(2) Sous échantillonnage des images	tâches et données	faible	élevé
(3) Détection des primitives	tâches et données	faible	faible
(4) Suivi de primitives	tâches et données	très élevé	très élevé
(5) Estimation du mouvement	aucun	faible	faible
(6) Filtrage du mouvement	aucun	faible	très faible

TAB. 2.9 – Caractéristiques des étapes de l’application de stabilisation.

Très classiquement, nous mettrons en oeuvre le parallélisme de données de ces étapes en travaillant sur des portions de l’image originale diffusées sur l’ensemble des processeurs au niveau MIMD et en utilisant ALTIVec pour exprimer le parallélisme de donnée SIMD au sein du code exécuté sur chaque processeur, tirant ainsi parti de l’efficacité d’ALTIVec lors du traitement de larges volumes de données.

2.7.3.1 Parallélisation des étapes de pré-traitement

Lors des étapes 1 et 2, la zone de détection de taille $q \times r$ est partagée entre les p processeurs disponibles, chacun travaillant alors sur une bande de taille $(\frac{q}{p} + 2T + k, r + 2T + k)$, la marge $2T + k$ étant nécessaire au suivi. Chacune de ces bandes contient alors une ou plusieurs bandes de détection.

2.7.3.2 Parallélisation des étapes de détection et suivi

La phase 3 est effectuée par chaque processeur p sur une bande de détection de taille $(\frac{q}{p}, r)$ et chaque processeur renvoie une liste de $\frac{n}{p}$ primitives qui seront suivies lors de l’étape 4. Le nombre de bandes est fonction du nombre de points que doit détecter chaque processeur à raison de 1 bande pour 3 points. Cette répartition est faite aussi équitablement que possible entre les processeurs, avec au plus

une différence de 1 point entre le processeur le plus chargé et le moins chargé. Par exemple pour $n = 84$, avec 8 processeurs, 4 processeurs détectent 11 points et 4 autres détectent 10 points. Avec 28 processeurs, ils ont tous exactement 3 points à traiter.

L'étape de suivi (4) est l'étape la plus coûteuse en temps de traitement. Elle autorise l'exploitation de deux niveaux de parallélisme. Chaque processeur recherche les $\frac{n}{p}$ motifs qu'il a précédemment détectés. Par ailleurs, les résultats de SSD sont obtenus en utilisant les instructions SIMD ALTIVec. De cette manière chaque processeur est capable de traiter jusqu'à 16 pixels en une seule opération, ce qui réduit d'autant le nombre d'opérations effectuées. Cette fonction de SSD existe en deux versions : une fonction simple, appliquée lors des deux premiers étages de la recherche multi-résolution travaille sur des nombres de type entier et une version plus complexe qui effectue des interpolations bilinéaires et travaille sur des nombres à virgule flottante au dernier niveau d'échelle.

Après la phase de suivi, toutes les listes d'appariement de points obtenues par chaque processeur sont concaténées en une seule liste. Ceci implique la collecte des résultats par un seul processeur (via la primitive MPI_gather) et représente l'essentiel des communications entre les processeurs. Finalement les étapes d'estimation du mouvement et de filtrage (étapes 5 et 6), implantées en séquentiel permettent de réaliser la stabilisation. La figure 2.12 résume le schéma de parallélisation de l'algorithme.

2.7.4 Performances

Les performances de l'algorithme de stabilisation implanté sur la machine BABYLON sont données dans les tableaux 2.10 à 2.12. Chaque tableau détaille pour un nombre de processeurs (P) allant de 1 à 28 les temps d'exécution en milliseconde de chaque étape de l'algorithme ($T_{1,2}, T_3, T_4$ et $T_{5,6}$), les gains respectifs de chaque étape par rapport à sa version séquentielle ($\gamma_{1,2}, \gamma_3$ et γ_4), les temps de communications (T_{comm}), le temps total d'exécution (T_{total}) et le gain total obtenu (Γ). Chaque tableau correspond à un format d'image allant de 640x480 à 5120x3840, en paramétrant l'algorithme pour rechercher 84 primitives à une distance T égale à 5% du format d'image. Ces mesures ayant été effectuées avant que C+FOX ne soit entièrement fonctionnel, nous avons donc simulé le *broadcast* Firewire en distribuant hors-ligne les séquences vidéos sur les disques locaux des nœuds du *cluster*.

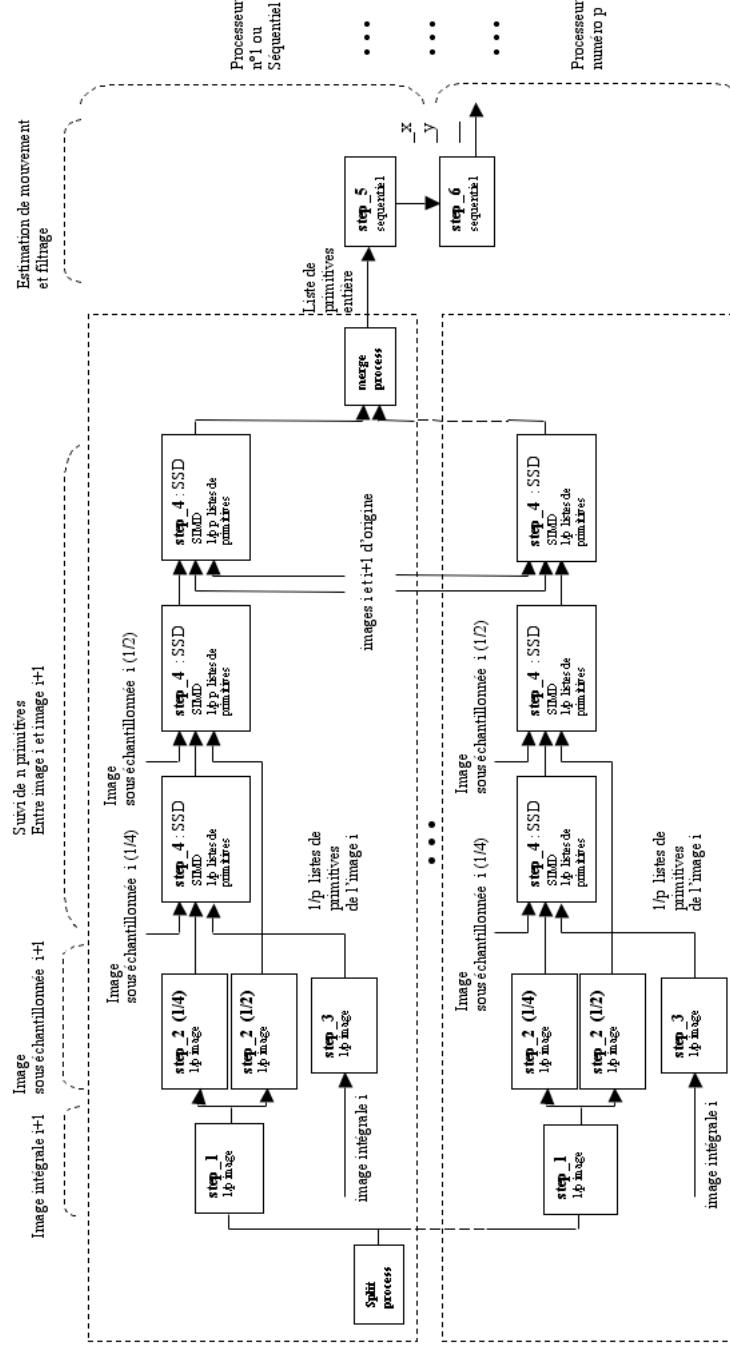


FIG. 2.12 – Synopsis de l’algorithme parallèle de stabilisation.

	$T_{1,2}$	$\gamma_{1,2}$	T_3	γ_3	T_4	γ_4	$T_{5,6}$	T_{comm}	T_{total}	Γ
P = 1	2,70	-	0,40	-	78,20	-	1,10	-	82,40	-
P = 2	1,70	1,59	0,20	2,0	3,60	21,72	1,10	0,10	6,60	12,48
P = 4	1,20	2,25	0,16	2,50	1,80	43,44	1,10	0,90	4,26	19,34
P = 8	1,0	2,70	0,12	3,33	1,0	78,20	1,10	0,60	3,22	25,59
P = 16	0,80	3,38	0,10	4,0	0,60	130,33	1,10	1,40	2,60	31,69
P = 28	0,80	3,38	0,06	6,67	0,40	195,50	1,10	2,0	2,36	34,92

TAB. 2.10 – Performances (en ms) de la stabilisation — Image 640x480

	$T_{1,2}$	$\gamma_{1,2}$	T_3	γ_3	T_4	γ_4	$T_{5,6}$	T_{comm}	T_{total}	Γ
P = 1	34,2	-	1,4	-	357,1	-	1,1	-	393,8	-
P = 2	20,2	1,69	0,4	3,5	21,8	16,38	1,1	0,1	43,5	9,05
P = 4	12,9	2,65	0,2	7,0	11,1	32,17	1,1	0,7	25,3	15,57
P = 8	9,4	3,64	0,2	7,0	5,9	60,53	1,1	1,0	16,6	23,72
P = 16	7,7	4,44	0,1	14,0	3,5	102,03	1,1	1,7	12,4	31,76
P = 28	6,8	5,03	0,1	14,0	2,0	178,55	1,1	2,2	10,0	39,38

TAB. 2.11 – Performances (en ms) de la stabilisation — Image 2560x1920

	$T_{1,2}$	$\gamma_{1,2}$	T_3	γ_3	T_4	γ_4	$T_{5,6}$	T_{comm}	T_{total}	Γ
P = 1	124,0	-	1,9	-	1226,0	-	1,1	-	1353,0	-
P = 2	72,0	1,72	0,5	3,80	80,0	15,33	1,1	0,4	153,6	8,81
P = 4	44,9	2,76	0,3	6,33	39,7	30,88	1,1	0,8	86,0	15,73
P = 8	32,4	3,83	0,2	9,50	21,7	56,50	1,1	0,9	55,4	24,42
P = 16	24,7	5,02	0,1	19,0	11,9	103,03	1,1	2,0	37,8	35,79
P = 28	21,4	5,79	0,1	19,0	6,2	197,74	1,1	4,5	28,8	46,98

TAB. 2.12 – Performances (en ms) de la stabilisation — Image 5120x3840

Plusieurs points sont à noter :

- Globalement, les gains mesurés en utilisant la totalité des processeurs de BABYLON sont très satisfaisants. Des gains de l'ordre de 35 à 50 sont atteints. Cette implantation parallèle est alors suffisamment performante pour rendre possible son exécution à la cadence de 25 images par secondes avec 2 processeurs pour des images 640x480, 4 processeurs pour les images 2560x1920 et 16 processeurs pour les images 5120x3840.
- De manière classique, les temps supplémentaires de synchronisation et de communication augmentent avec le nombre de processeurs. Mais dans le cas des images de grande taille, quel que soit le nombre de processeurs, les temps de traitement restent suffisamment élevés pour compenser les coûts de ces communications. Pour les images de petite taille, la situation est différente. Les temps de communication pour 28 processeurs (environ 2 ms) deviennent trop importants pour être compensés par les temps de traitement.
- Les étapes 1 et 2 subissent une accélération relativement faible quel que soit le nombre de processeurs et la taille de l'image. Ceci s'explique par la présence du coefficient $2T + k$ — indépendant du nombre de processeurs — dans l'expression de la taille du bloc de données traité par chaque processeur.
- L'étape 4 subit l'accélération la plus forte — entre 15,33 et 197,74 — quel que soit la taille de l'image. Comme prévu, ce gain très important s'explique par le recours simultané à MPI et à l'extension ALTIVec. Comme cette étape correspondait à plus de 90% du temps d'exécution globale, le gain final dépend en grande partie des performances de cette parallélisation MIMD-SIMD.

2.7.5 Conclusion

Au final, les performances de l'application de stabilisation permettent de mettre en avant les bénéfices de l'architecture hybride mise en place au sein de BABYLON. Il est possible d'obtenir des gains de grande amplitude pour des applications de vision artificielle de complexité moyenne en utilisant un nombre de noeuds restreint, en tirant parti des trois niveaux de parallélisme offerts par la machine.

Pour cette application, tous les cas de figures expérimentaux montrent que le gain fourni par BABYLON est au minimum le double du gain maximal théorique d'une machine MIMD construites à partir de processeurs sans extension SIMD. L'adéquation de la couche SIMD fournie par ALTIVEC aux traitements bas niveau des images confirme, comme nous l'avions supposé, son intérêt au sein de notre architecture.

Dans la section suivante, nous allons chercher à quantifier de manière analytique l'apport de ces trois niveaux de parallélisme en proposant un modèle adapté à ce type d'architecture hybride. Ce modèle nous permettra par la suite d'évaluer et d'analyser plus finement les performances des applications que nous implémenterons.

2.8 Modèle de performance

Pouvoir prédire les performances d'un algorithme parallèle permet de déterminer si le processus effectif de parallélisation vaut la peine d'être mené. Ces prédictions permettent aussi, *a posteriori*, de mettre en avant les causes probables des éventuelles faibles performances d'un tel algorithme. Dans le cadre des machines MIMD classiques, plusieurs modèles de performances ont été proposés. Mais ces modèles ne prennent pas en compte les spécificités d'une architecture hybride MIMD-SIMD. Nous allons donc voir comment, à partir de ces modèles simples, il est possible de proposer un modèle de performances adaptés aux architectures hybrides telle que BABYLON .

2.8.1 Loi d'Amdahl

La parallélisation d'algorithmes a pour but de réduire le temps d'exécution de l'algorithme séquentiel initial. Dans cette optique, on définit **l'accélération** comme étant le rapport entre le temps d'exécution séquentiel et le temps d'exécution parallèle. En général, trois types de temps interviennent dans cette expression :

- le temps de calcul purement séquentiel;
 - le temps de calcul potentiellement parallélisable;
 - le temps de communication dans la version parallèle.
-

Si l'on considère l'accélération $\psi(n, p)$ due à la parallélisation d'un problème de taille n sur p processeur, on obtient :

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

où $\sigma(n)$ représente le temps d'exécution purement séquentiel, $\varphi(n)$ le temps de calcul potentiellement parallélisable et $\kappa(n, p)$ le temps des communications dans la version parallèle. Comme $\kappa(n, p) \geq 0$, on obtient alors la relation suivante :

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

Soit f la fraction de calcul purement séquentiel :

$$f = \frac{\sigma(n)}{\sigma(n) + \varphi(n)}$$

Il vient :

$$\psi(n, p) \leq \frac{1}{f + (1 - f)/p}$$

Cette relation est appelée loi d'Amdahl [14]. Elle est basée sur l'hypothèse que, pour un problème de taille fixe, nous tentons de diminuer le temps nécessaire à sa résolution en augmentant le nombre de processeurs. Elle permet d'estimer la borne supérieure de l'accélération atteignable en utilisant un nombre de processeur de plus en plus grand.

2.8.2 Loi de Gustafson-Barsis

La loi d'Amdahl fait l'hypothèse que le but final du processus de parallélisation est de diminuer le temps d'exécution. Pour cela, on considère la taille du problème comme fixe et on augmente le nombre de processeurs. Il est intéressant d'inverser l'optique du problème, par exemple en prenant un algorithme dont la parallélisation ne va pas réduire le temps d'exécution mais augmenter la précision. Dans ce type de problème, nous fixons le temps d'exécution et laissons la taille du problème augmenter en fonction du nombre de processeur. Dans ce cas, posons s comme étant la fraction de temps parallèle passée à effectuer des calculs séquentiel :

$$s = \frac{\sigma(n)}{\sigma(n) + \varphi(n)/p}$$

La quantité $(1 - s)$ représente alors la fraction de temps parallèle passée à effectuer des calculs parallèles :

$$(1 - s) = \frac{\varphi(n)/p}{\sigma(n) + \varphi(n)/p}$$

Il vient donc :

$$\sigma(n) = (\sigma(n) + \varphi(n)/p)s$$

$$\varphi(n) = (\sigma(n) + \varphi(n)/p)(1 - s)p$$

On obtient alors

$$\psi(n, p) \leq p + (1 - p)s = s + (1 - s)p$$

Cette équation définit la loi de Gustafson-Barsis [90] qui permet d'évaluer l'accélération d'un programme parallèle exécuté sur des problèmes dont la taille dépend du nombre de processeur [170].

2.8.3 Modèle de performance hybride

Les modèles proposés par Amdahl et Gustafson ne tiennent pas compte de l'accélération fournie par des processeurs équipés d'extensions SIMD. Un nouveau modèle a été proposé par Leo Chin Sim, Heiko Schröder et Graham Leedham [159] pour intégrer cette composante au sein d'un modèle de performance hybride. Ce modèle hybride est basé sur la loi de Gustafson mais y intègre une contribution des systèmes SIMD. Pour ce faire, il faut reprendre la définition initiale de l'accélération.

$$\psi(n, p, x) \leq \frac{\sigma(n) + \varphi(n) + \chi(n)}{\sigma(n) + (\varphi(n) + \chi(n)/x)/p}$$

Dans cette équation, le terme SIMD est introduit par $\chi(n)$ et x qui représentent respectivement le temps de calcul séquentiel potentiellement parallélisable via le mode SIMD et le gain fourni par le mode SIMD. En reprenant les définitions mises en place par Gustafson, on pose :

$$\alpha = \frac{\sigma(n)}{\sigma(n) + (\varphi(n) + \chi(n)/x)/p}$$

$$\beta = \frac{\varphi(n)/p}{\sigma(n) + (\varphi(n) + \chi(n)/x)/p}$$

$$\gamma = \frac{\chi(n)/xp}{\sigma(n) + (\varphi(n) + \chi(n)/x)/p}$$

où α , β et γ représentent respectivement la fraction de code purement séquentiel, la fraction de code parallélisable en MIMD et la fraction de code parallélisable en SIMD. Au final, on obtient :

$$\psi(n, p, x) \leq \alpha + p\beta + px\gamma$$

La question se pose de savoir si cette nouvelle formulation respecte les modèles classiques et se comporte correctement dans un certains nombre de cas simples. On retrouve par exemple l'expression de l'accélération d'une machine MIMD classique de manière naturelle en posant $\gamma = 0$. Dans ce cas, on retrouve une expression de l'accélération qui n'utilise pas de le gain SIMD x et que l'on peut comparer à celle donnée par la loi de Gustafson-Barsis en posant $s = \alpha$:

$$\psi(n, p, x) \leq \alpha + p(1 - \alpha)$$

Un autre cas intéressant est le cas où $\alpha = \beta = 0$. Dans cette configuration, nous évaluons le gain d'une machine MIMD-SIMD parfaite dans laquelle le parallélisme est total. On obtient alors le gain maximum théorique d'une machine hybride :

$$\psi(n, p, x) \leq npx$$

2.8.4 Mise en oeuvre

Nous allons confronter les mesures au modèle théorique présenté dans la section précédente afin de juger de la pertinence de ce dernier. Les tableaux 2.13 à 2.15 reprennent les résultats des mesures effectuées en indiquant : le nombre de processeurs (p), le facteur de gain SIMD (x) mesuré expérimentalement sur notre application, les valeurs des coefficient α , β et γ , le gain prédit par le modèle hybride, le gain effectivement mesuré et le pourcentage d'erreur entre le gain mesuré et le gain du modèle hybride.

p	x	α	β	γ	$\psi(n, p, x)$	Mesure	Erreur
2	11,1	0,3	0,22	0,48	11,39	12,48	-9,51%
4	11,1	0,38	0,25	0,37	17,80	19,34	-8,60%
8	11	0,47	0,26	0,27	26,31	25,59	2,73%
16	11	0,65	0,2	0,15	30,25	31,69	-4,76%
28	10,9	0,72	0,19	0,09	33,50	34,92	-4,21%

TAB. 2.13 – Évaluation du modèle de performance hybride - Image 640x480

p	x	α	β	γ	$\psi(n, p, x)$	Mesure	Erreur
2	10,5	0,04	0,46	0,4	9,36	9,05	3,31%
4	9,3	0,08	0,55	0,37	16,04	15,57	2,95%
8	7,1	0,13	0,53	0,34	23,68	23,72	-0,16%
16	5,7	0,2	0,55	0,25	31,8	31,76	0,12%
28	4,3	0,28	0,56	0,16	35,22	39,38	-11,79%

TAB. 2.14 – Évaluation du modèle de performance hybride - Image 2560x1920

p	x	α	β	γ	$\psi(n, p, x)$	Mesure	Erreur
2	8,1	0,01	0,47	0,5	9,05	8,81	2,65%
4	7,7	0,02	0,52	0,46	16,26	15,73	3,30%
8	6,8	0,03	0,58	0,39	25,88	24,42	5,65%
16	5,5	0,08	0,62	0,3	36,4	35,79	1,67%
28	5,3	0,17	0,65	0,18	45,08	46,98	-4,21%

TAB. 2.15 – Évaluation du modèle de performance hybride - Image 5120x3840

Ces mesures montrent que le modèle hybride permet d'obtenir des prédictions très proche du gain effectivement mesuré, avec une erreur maximale de l'ordre de 10%. L'origine de cette erreur réside principalement dans le fait que la mesure du gain SIMD x est relativement imprécise, l'accélération fournie par ALTIVec étant sujette à de grandes variations. En outre, l'estimation d'un gain SIMD composite, c'est-à-dire mettant en oeuvre des types de données vectorielles différents est relativement complexe. Ces imprécisions sur la valeur de x se répercutent ainsi sur la valeur finale de $\psi(n, p, x)$. Un autre facteur d'erreur provient du fait que le modèle utilisé ne tient pas compte des temps de communications. Néanmoins, cette marge d'erreur est suffisamment faible pour nous permettre d'utiliser ce modèle afin de prédire le comportement des applications que nous serons amenés à développer sur la machine BABYLON. Nous verrons au chapitre 6 comment ce modèle se comporte avec d'autres types d'applications de complexité supérieure.

2.9 Conclusion

BABYLON est une architecture de type *cluster* novatrice de par ses trois niveaux de parallélisme et ses deux réseaux de communications dédiés qui limitent les pertes de performances qui surviennent dans des architectures plus classiques.

La programmation de cette architecture avec des outils standards autorise l'exploitation efficace des ressources de cette machine : MPI, pThread et ALTIVEC permettent de tirer parti avantageusement des trois niveaux de parallélisme fournis par la machine et C+FOX assure la qualité et la vitesse d'acquisition des données vidéos. Les résultats exhibés par l'application de stabilisation d'images confirment les choix technologiques effectués avec des résultats très satisfaisants en termes de gain et de performance finale. Néanmoins, nous ne remplissons pour l'instant qu'une partie du cahier des charges que nous nous sommes fixé au chapitre précédent. En effet, si l'utilisation d'outils comme MPI, pThread ou ALTIVEC reste abordable pour la vectorisation ou la parallélisation d'applications simples, elle devient prohibitive lorsque la complexité des applications augmente. Dans le cas de l'application de stabilisation par exemple, la création du schéma de communication et de la stratégie de parallélisation présentées sur la figure 2.12 représente près de 40% du code final. Son adaptation en vue de gérer des images d'entrées de tailles diverses, un nombre de processeurs variables ou un nombre de primitives variables complexifie largement l'écriture du code MPI résultant. Concernant ALTIVEC, son utilisation au sein d'algorithmes de faible complexité est relativement aisée mais la vectorisation d'algorithmes mettant en jeu plusieurs types de données ou plusieurs schémas de vectorisation se révèle en pratique assez complexe. Un exemple flagrant est par exemple la fonction de calcul de la SSD sur les images sous-échantillonnées. Le code vectoriel de cette fonction représente près de 150 lignes. Le développement de telles fonctions est long et sujet aux erreurs du fait de la quantité de paramètres à prendre en compte, comme le niveau de déroulage, les opérations de «prefetch» ou les opérations de transtypage. En outre, l'écriture optimale de ces fonctions vectorisées n'est pas triviale à mettre en oeuvre et peut conduire, si ces optimisations ne sont pas effectuées, à des performances très en deçà du maximum théorique. L'annexe III présente un extrait de code emblématique du saut de complexité que nécessite l'utilisation efficace d'ALTIVEC.

Si nous voulons répondre à notre cahier des charges initial, il est nécessaire d'utiliser des outils de plus haut niveau qui nous permettront de développer de manière rapide et efficace des applications de vision de plus grande complexité tout en limitant les difficultés de mise au point. Ces outils devront cibler en particulier les problématiques de calcul vectoriel via ALTIVEC, la définition d'applications MIMD via MPI et pThread. Ils devront fournir par ailleurs une interface permettant de fournir un niveau de programmabilité élevé pour BABYLON tout en conservant les performances mises en évidence dans ce chapitre.

Chapitre 3

Outils logiciels pour le calcul parallèle

«*When we had no computers, we had no programming problem either.*

When we had a few computers, we had a mild programming problem.

Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.»

Edsger Dijkstra

Le chapitre précédent a montré qu'il est possible, avec des outils comme ALTIVEC ou MPI, de tirer parti efficacement de l'architecture de *cluster* hybride MIMD-SIMD de BABYLON. Néanmoins, chacun de ces outils nécessite une prise en main non-triviale si l'on désire utiliser tout leur potentiel. Ainsi, comme nous l'avons vu au paragraphe 2.7.3, si les problèmes simples et réguliers se vectorisent de manière quasi-naturelle grâce à l'interface C de l'extension ALTIVEC, peu d'algorithmes conservent une écriture simple une fois vectorisés efficacement. En effet, les divers optimisations et pré-traitements [11, 52] nécessaires pour assurer de bonnes performances finissent souvent par représenter une large partie du code final, masquant le noyau applicatif de l'algorithme. Pour pThread, son interface de très bas niveau souffre des mêmes lacunes que celle d'ALTIVEC même si des sur-couches orientées objet permettent de réduire les difficultés d'utilisation. Dans le cas de MPI, la difficulté principale provient de la mise au point du schéma de communication et de la répartition des différentes tâches constituant l'algorithme. Cette étape de mise au point est critique car elle conditionne les performances finales de l'algorithme qui dépendent entièrement du ratio calcul/communication. Elle est aussi la première source d'erreurs – comme les inter-blocages – qui ralen-

tit d'autant le développement d'applications MIMD.

Un niveau supplémentaire de complexité est franchi avec BABYLON où l'utilisation conjointe d'ALTIVec, de pThread et de MPI est quasi-obligatoire. Pour l'utilisateur final, souvent peu éclairé sur l'ensemble des difficultés exposées précédemment, le portage efficace d'un code existant sur cette architecture hybride constitue dès lors un challenge non-négligeable. Comme nous l'avons vu au paragraphe 2.7, une approche efficace consiste à découpler les problématiques de parallélisation en effectuant tout d'abord une parallélisation MIMD et SMP du problème suivi d'une phase de vectorisation des éléments séquentiels restant. Malheureusement, le faible niveau d'abstraction des outils rend très difficile l'écriture et la maintenance d'un tel code hybride. Il est donc nécessaire de se tourner vers des outils de plus haut niveau. Ces outils devront répondre aux critères suivants :

- **L'efficacité**, qui décrit leur capacité à produire un code exécutable dont les performances sont les plus proches possibles de celles d'un code écrit et optimisé «à la main».
- **L'expressivité**, qui décrit leur capacité à décrire toutes les situations envisageables.
- **L'accessibilité**, qui décrit leur capacité à être utilisé avec un minimum de prise en main, à permettre l'intégration de code existant et la réutilisation de principes de développement connus et maîtrisés.

Dans le cadre de nos travaux, nous allons définir des modèles de programmation et des outils qui permettront d'exploiter les différents niveaux de parallélisme de notre architecture et un niveau d'abstraction supérieur à celui existant afin de simplifier leur utilisation conjointe. Ces outils doivent mettre l'accent sur l'efficacité — afin de permettre de conserver un niveau de performances de BABYLON comparable à celui obtenu au paragraphe 2.7 — et l'accessibilité — afin de laisser aux développeurs issus de la communauté Vision la possibilité de réutiliser des modèles, langages et bibliothèques qui leur sont familiers. L'expressivité n'est bien sûr pas reléguée au second plan et contribue pour une large part à faire accepter les outils de développement disponibles sur BABYLON aux développeurs d'applications de vision par ordinateur. Les sections suivantes dressent un panorama des différentes solutions disponibles et présentent les avantages et inconvénients respectifs de ces dernières.

3.1 Modèles et outils pour la programmation SIMD

Très peu d'outils ont été proposés pour éléver le niveau d'abstraction offert par l'API de l'extension ALTIVEC telle que nous l'avons décrite au paragraphe 2.6. En particulier, aucun véritable modèle de programmation haut niveau n'a été proposé, et ce bien qu'un modèle basé sur la manipulation de tableaux numériques s'adapte intuitivement à l'interface C d'ALTIVEC. Les principaux outils disponibles sont l'*Apple Accelerate.Framework* et VAST.

3.1.1 L'*Apple Accelerate.Framework*

Apple fournit un ensemble de bibliothèques [46] basées sur ALTIVEC qui comble une partie des manques de l'API de base de l'extension. Parmi ces bibliothèques on trouve :

- Une implantation vectorielle des bibliothèques d'algèbre linéaire BLAS et LAPACK [18];
- **vDSP** : une collection de fonctions de traitement du signal comme des transformées de Fourier rapides ou des convolutions;
- **vImage** : un ensemble de fonctions de traitement d'images comme la mise à l'échelle, la rotation, la convolution, des opérations sur les histogrammes et des routines de conversions de formats de pixels;
- **vMathLib** : des fonctions mathématiques de base avec une précision élevée : division, fonctions trigonométriques et exponentielles;
- **vBasicOps** : une collection de fonctions comblant les lacunes de l'API d'ALTIVEC concernant les types entiers (multiplication, division, etc...);
- **vBigNum** : des fonctions de traitements d'entiers 256, 512 et 1024 bits ainsi que des fonctions de base de cryptographie.

Le spectre des fonctions proposées est très large et l'implémentation est de très bonne qualité. Malheureusement, le niveau de programmation reste très bas et se base sur une manipulation directe de tableaux C. En outre, aucune optimisation inter-appel n'est effectuée, rendant la composition de ces fonctions peu efficace.

3.1.2 VAST

VAST est un outil commercial de pré-traitement et de réécriture de code [3] ciblant différentes plateformes matérielles. Il est en outre compatible avec les machines équipées d'extension ALTIVec et est capable de réécrire dans une large mesure un code C classique en code C ALTIVec. Cette réécriture permet de vectoriser automatiquement des boucles, des branchements conditionnels, de détecter le parallélisme lié à certaines structures de données. Au final, le code résultant est plus efficace, exhibant un gain de l'ordre de 75 à 90 pour cent du gain théorique maximal fourni par ALTIVec.

Malgré ses performances élevées et sa facilité d'utilisation, VAST reste un outil professionnel coûteux et dont la maîtrise n'est pas immédiate. En outre, bien que le spectre de constructions «vectorisables» par VAST soit large, il est parfois nécessaire de reformuler de manière non triviale certaines portions de code afin que leur transformation soit optimale.

3.2 Modèles et outils pour la programmation MIMD

La programmation de machines MIMD — et plus particulièrement des machines MIMD à mémoire distribuée — a fait l'objet de nombreuses études et, contrairement à ALTIVec, de nombreux modèles de programmation ont été proposés. Une classification englobant — entre autre — ces modèles a été proposée par Skillicorn et Talia [164] et est fondée sur des éléments comme la forme implicite, explicite ou semi-implicite du parallélisme mis en œuvre, des étapes de communications, de placement et de synchronisations des processus au sein du modèle et le degré de contrôle de l'utilisateur sur ces étapes.

Sans revenir à une analyse détaillé de l'ensemble des modèles MIMD-DM présentés dans cette classification, nous nous focaliserons sur les modèles dit implicites ou semi-implicites car ils répondent par définition aux objectifs que nous nous sommes fixés en terme d'expressivité et d'accessibilité, en masquant les étapes bas-niveaux d'exploitation du parallélisme. Ils sont souvent considérés comme étant les plus à même de fournir un cadre de développement simple et efficace. Parmi ces modèles, les approches les plus marquantes sont les squelettes algorithmiques (paragraphe 3.2.1) et les *Design Pattern* (paragraphe 3.2.2) bien que d'autres approches aient amené leur contribution à ce domaine (paragraphe 3.2.3).

3.2.1 Les squelettes algorithmiques

Le concept de squelette algorithmique [45, 163] est basé sur le fait que la mise en œuvre du parallélisme dans les applications se fait très fréquemment en utilisant un nombre restreint de schémas récurrents. Ces schémas explicitent les calculs menés en parallèle et les interactions entre ces calculs. La notion de squelette correspond alors à tout ce qui, dans ces schémas, permet de contrôler les activités de calcul. Un squelette algorithmique prend donc en charge un schéma de parallélisation précis. Ainsi, toutes les opérations bas-niveau nécessaires à la mise en œuvre de la forme de parallélisme choisie sont contenues dans le code du squelette. L'utilisateur paramètre ce squelette en fournissant les fonctions de calcul de son algorithme. Ces fonctions seront alors exécutées en parallèle et les données transiteront entre elles selon le schéma que représente le squelette.

Les squelettes algorithmiques rendent donc compte d'une volonté de structurer la programmation parallèle comme le fut la programmation séquentielle [43]. La complexité de la programmation parallèle étant alors restreinte par la limitation délibérée de l'expression du parallélisme à des formes clairement identifiées et souvent dépendantes du domaine d'application considéré [87].

Parmi les projets basés sur ces modèles, nous citerons en particulier P3L, eSkel, MUESLI, Skipper et Muskel.

3.2.1.1 P3L — Pisa Parallel Programming Language

Le *Pisa Parallel Programming Language* [22] – ou P3L – est un langage de programmation parallèle basé sur le concept des squelettes algorithmiques. P3L utilise du code C comme code séquentiel et possède une syntaxe proche du C pour la spécification des structures parallèles. La conception d'une application P3L passe donc par deux étapes : la spécification en C des tâches séquentielles et l'écriture en P3L de la structure à base de squelettes. Un compilateur spécifique [41] produit ensuite un exécutable unique à partir de ces deux sources.

Un exemple de code P3L est donné dans le listing 3.1. Les étapes du «pipeline» sont définies par leur prototype P3L et leur code C (lignes 1–2). Le squelette «pipeline» est défini à la ligne 4 en spécifiant la succession des étapes et leurs entrées/sorties.

L'intérêt de P3L est alors de conserver une définition simple des tâches sé-

quentielles sous forme de code C et de déléguer à un outil externe le fait de générer le code exécutable à partir de la description de l'application.

Listing 3.1 – Déclaration d'un pipeline avec P3L

```

1 seq stage1 in (float x) out (int y) ... end seq
2 seq stage3 in (int x) out (float y) ... end seq
3 seq stage2 in (float x) out (float y) ... end seq
4
5 pipe main in(float x) out(float y)
6 stage1 in(x) out (int z)
7 stage2 in(z) out (float w)
8 stage2 in(w) out (y)
9 end pipe

```

3.2.1.2 eSkel

eSkel [44] est une bibliothèque C basée sur MPI [43] qui propose une implantation des squelettes algorithmiques via un ensemble de fonctions encapsulant les schémas de communication d'un nombre restreint de squelettes. eSkel propose en outre des moyens de spécifier des communications *ad hoc* entre processus et un large panel d'options fournissant une palette de variations sur les squelettes de bases.

Le listing 3.2 décrit ainsi un pipeline à trois étapes exprimé avec l'API de eSkel. Pour ce faire, il est nécessaire de déclarer la fonction C séquentielle qui servira de code pour les étapes du pipeline (ligne 1), un certain nombre de paramètres spécifiant comment les données d'entrées et de sorties seront réparties sur les différents nœuds (lignes 15–21). Enfin, l'appel à la fonction Pipeline lance l'exécution du squelette suivant les options précisées dans ses paramètres.

On notera que l'interface de ESkel reste très proche de celle de MPI – utilisant des pointeurs de fonction C et des tableaux. Cette interface permet au développeur MPI de s'imprégner facilement des concepts des squelettes tout en gardant une certaine familiarité avec leur environnement. Par contre, cette facilité d'accès est contrebalancée par le niveau relativement bas de l'interface et la verbosité du code ainsi produit.

Listing 3.2 – Déclaration d'un pipeline avec ESkel

```

1 eSkel_molecule_t* f(eSkel_molecule_t *in) { ... }

2

3 int main (int argc, char *argv[])
4 {
5     int i, mystagenum, outmul;
6     int inputs[8], results[8];
7     eSkel_molecule_t *(st[3])(eSkel_molecule_t *);
8     spread_t      spreads[3];
9     MPI_Datatype   types[3];
10    Imode_t       imodes[3];

11
12    MPI_Init(&argc, &argv);
13    SkelLibInit();

14
15    for (i=0; i<3; i++)
16    {
17        spreads[i] = SPGLOBAL;
18        types[i]   = MPI_INT;
19        imodes[i]  = IMPL;
20        st[i]      = &f;
21    }

22
23    Pipeline(3,imodes,st,myrank(),BUF,spreads,
24              types,inputs,2,4,results,INPUTSZ,
25              &outmul,8,mycomm());
26
27    MPI_Finalize();
28    return 0;
29 }
```

3.2.1.3 Skipper

SKiPPER [158, 157] est un environnement de programmation dédié au prototypage rapide d'applications parallèles pour la vision artificielle via l'utilisation de squelettes algorithmiques [87, 47]. SKiPPER propose un ensemble de squelettes orientés vision et qui s'expriment comme des fonctions d'ordre supérieur au sein d'un langage fonctionnel proche de ML. SKiPPER permet de spécifier un algorithme parallèle via la composition de ces fonctions, chacune de ces fonctions

prenant alors en paramètres des fonctions séquentielles écrites en C. SKiPPER transforme cette spécification en un graphe de tâches dont les nœuds représentent les fonctions séquentielles ou les processus de contrôle des squelettes et dont les arêtes représentent les communications.

3.2.1.4 MUESLI

MUESLI [119] est une bibliothèque orientée objet écrite en C++ et qui est basée sur un modèle qui définit l'exécution parallèle comme étant une séquence de tâches parallèles. Chacune de ces tâches peut contenir d'autres squelettes parallèles ou utiliser des primitives dédiées au parallélisme de données. L'implémentation de MUESLI repose sur la définition d'une hiérarchie de classes encapsulant le concept de tâche séquentielle. Ces tâches sont ensuite passées en argument à des instances de classes implantant différents squelettes algorithmiques. Cette approche orientée objet permet facilement de réutiliser du code existant sous la forme de classe ou de fonction C ou C++ et fournit un bon niveau d'expressivité. Néanmoins, le surcoût engendré par cette méthode limite les performances de MUESLI. Un exemple de déclaration de squelette utilisant MUESLI est donné sur le listing 3.3 qui définit un pipeline de trois étapes. Les fonctions séquentielles sont définies aux lignes 1–3. Ces fonctions sont alors insérées au sein d'objets fournis par MUESLI (lignes 5–7). Une fois instanciés, ces objets sont passés en paramètre au constructeur de la classe Pipe qui encapsule le comportement du squelette «pipeline». L'appel de la méthode start déclenche l'exécution du programme.

Listing 3.3 – Déclaration d'un pipeline avec MUESLI

```

1 int* func_stage1();
2 int* func_stage2(int in);
3 void func_stage3(int in);
4
5 Initial<int> stage1(func_stage1);
6 Atomic<int, int> stage2(func_stage2);
7 Final<int> stage3(func_stage3);
8 Pipe app(stage1, stage2, stage3s);
9 app.start();

```

3.2.1.5 Muskel

Muskel [54] est une bibliothèque JAVA basée sur les squelettes algorithmiques. Le choix de JAVA comme langage cible est motivé par sa portabilité et son aspect orienté objet. Un exemple d’application écrite avec Muskel est donné dans le listing 3.4.

Listing 3.4 – Déclaration d’un pipeline avec Muskel

```

1  class stage1 { ... };
2  class stage2 { ... };
3  class stage3 { ... };
4
5  Skeleton main = new Pipeline(new stage1,
6                      new Pipeline(new stage2,
7                                new stage3));
8
9  Manager manager = new Manager();
10 manager.setProgram(main);
11 manager.setContract(new ParDegree(10));
12 manager.setInputStream(inFile);
13 manager.setOutputStream(outFile);
14 manager.eval();

```

Dans cet exemple, un squelette «pipeline» est instancié à la ligne 5. Ensuite, la classe de gestion d’applications est créée. On lui fournit alors le squelette à exécuter, le nombre de nœuds à utiliser et les flux d’entrées/sorties nécessaires. L’appel à la méthode eval lance l’exécution du programme parallèle.

3.2.2 Les *Design Patterns*

les *Design Patterns* [8, 9] sont un concept particulièrement riche et qui possède de nombreux domaines d’application. Les *Design Patterns* permettent en effet de capitaliser l’expérience des concepteurs les plus expérimentés et ainsi d’éviter de «réinventer la roue». Tout d’abord restreints aux problématiques de la programmation séquentielle [84], plusieurs travaux de recherches ont proposé d’appliquer ce concept à la programmation parallèle [130, 53, 26, 160]. Les Design Patterns parallèles sont présentés comme des abstractions modélisant des schémas de parallelisation récurrents, d’une façon similaire aux squelettes algorithmiques.

Même si les squelettes algorithmiques ont été développés de manière complètement indépendante des *Design Patterns*, leur philosophie est très proche de celle qui sous-tend ces derniers. Néanmoins, les deux approches présentent des différences dont les points essentiels sont leur expressivité, la manière dont ils sont implantés, mais aussi les possibilités d'extensibilité. M. Danelutto expose ainsi les similitudes et différences notables entre les deux approches [53].

	<i>Design Patterns</i>	Squelettes algorithmiques
Type de programmation	Parallèle et distribuée	Parallèle
Niveau d'abstraction	Elevé	Très élevé
Imbrication	Possible	Possible
Réutilisation du code	Encouragée	Encouragée
Langage support	Orienté-objet de préférence	Tout langage
Extensibilité	Très élevée	Limitée

TAB. 3.1 – Comparaison des approches *Design Patterns* et Squelettes algorithmiques [53].

Parmi les projets de définitions *Design Pattern* parallèle nous citerons en particulier DPnDP et CO₂P₃S.

3.2.2.1 DPnDP

DPnDP [161] propose un ensemble d'outils et de bibliothèques permettant de définir des applications à base de *Design Patterns* parallèles en C++. Son modèle de programmation est basé sur la définition de graphes de tâches orientés. Les relations entre les nœuds de ces graphes sont alors explicitées par des *Design Patterns* spécifiques qui permettent de définir des stratégies de communication *ad hoc* pour implanter le schéma de parallélisation désiré. Pour ce faire, DPnDP utilise sa propre bibliothèque de passage de message appelée «*Node Layer*» qui effectue des communications nœud à nœud de manière à tirer partie des propriétés du graphe de tâches défini par l'utilisateur.

3.2.2.2 CO₂P₃S

CO₂P₃S [152] propose un ensemble d'outils permettant le développement de programmes parallèles en JAVA en utilisant un modèle basé sur les *Design patterns*. Son apport principal est de fournir des patrons génériques qui prennent en compte dans leurs paramètres des détails spécifiques à l'application [174]. Il propose aussi un outil – Meta-CO₂P₃S – qui permet à l'utilisateur de définir de

nouveaux patrons et de les inclure au sein du système principal. Pour définir une application parallèle sous CO₂P₃S, il suffit de choisir un patron, de spécifier ses paramètres et de lui fournir un ensemble de codes séquentiels qui viendront s'insérer aux emplacements adéquats au sein du patron.

3.2.3 Autres approches

Parmi les autres outils proposés, on trouve un ensemble de projets basés sur des modélisation semi-explicites des schémas de communications au sein d'un langage hôte. Parmi ces outils, on trouve par exemple EDEN et BSMLlib.

3.2.3.1 EDEN

EDEN [31] est un langage fonctionnel parallèle basé sur le langage Haskell [107]. Reposant sur les principes du λ -calcul, EDEN fournit une panoplie de constructions permettant de contrôler très finement le grain de la parallélisation d'une application tout en masquant au développeur les détails d'implémentation des communications. Bien que s'appuyant sur le principe de l'évaluation paresseuse¹ de Haskell, EDEN est capable de s'en abstraire afin d'implanter ses primitives parallèles. Ecrire un programme parallèle avec EDEN passe par la transformation de fonctions séquentielles en processus parallèles et l'instanciation de tels processus. Pour cela, EDEN propose deux construction :

- `process :: (Trans a, Trans b) => (a -> b) -> Process a b` qui effectue la transformation d'une fonction de type `(a -> b)` en un processus abstrait.
- `(#) :: (Trans a, Trans b) => Process a b -> a->b` qui effectue l'instanciation et donc l'exécution de l'activité parallèle associée à un processus abstrait.

Plusieurs extensions sont développées pour EDEN, dont en particulier une bibliothèque [124] introduisant un ensemble de constructions parallèles récurrentes au sein du langage, fournissant ainsi à EDEN un équivalent des squelettes algorithmiques.

¹L'évaluation paresseuse ou évaluation retardée est une technique de programmation où le programme n'exécute pas de code avant que les résultats du code ne soient réellement nécessaires.

3.2.3.2 BSMLlib

BSMLlib [125] est une extension de ML pour la programmation fonctionnelle d’algorithmes parallèles suivant le modèle du *Bulk Synchronous Parallelism* [182] (BSP) en mode direct. L’exécution d’un programme BSP est une séquence de super-étapes. Chaque super-étape est divisée en trois phases successives et logiquement disjointes : une phase de calcul local, une phase d’échange de données entre processus et finalement une barrière de synchronisation globale. A l’issue de cette barrière de synchronisation globale, les données échangées sont effectivement disponibles pour la super-étape suivante (figure 3.1).

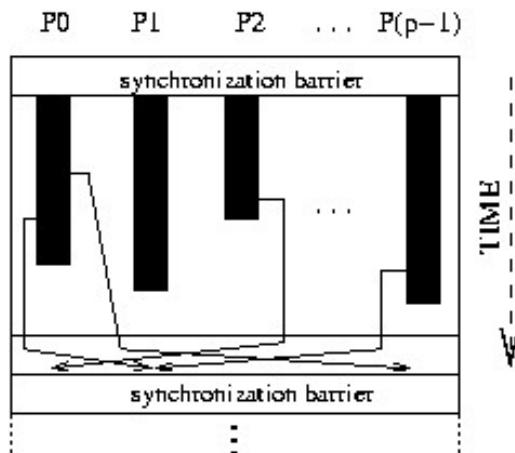


FIG. 3.1 – Déroulement d’un programme parallèle utilisant le modèle BSP [97]

BSMLlib implante les primitives BSP en Objective Caml avec la bibliothèque MPI à l’aide d’un ensemble de primitives issues d’un calcul confluant, le BS λ -calcul [126].

3.3 Discussion

A la lumière des contraintes présentées précédemment, les différentes solutions présentées ici peuvent être classées entre approches basées sur des nouveaux langages, sur des outils de générations de code ou sur des bibliothèques externes.

3.3.1 Approches basées sur des nouveaux langages

Ces solutions, comme P3L ou EDEN par exemple, apportent efficacité et expressivité car elles mettent en oeuvre des méthodes et des modèles de programma-

tion en adéquation avec la problématique de la programmation parallèle et du type d'outil utilisé. Mais, si l'on considère les critères mis en avant au début de ce chapitre, la création d'un ou plusieurs nouveaux langages et leur intégration dans le cycle de développement d'applications compromet notre objectif d'accessibilité. En outre, de tels langages ont souvent une durée de vie relativement courte du fait d'un support et d'une portabilité plus difficile à assurer, d'un manque d'outils de développement avancés (comme un débogueur par exemple) et du fait que l'utilisation de plusieurs de ces langages spécifiques au sein d'un même programme est souvent complexe voire impossible.

3.3.2 Approches basées sur des générateurs de code

Des approches comme VAST, DPnDP ou CO₂P₃S se révèlent performantes et apportent un bon niveau d'expressivité. Souvent basées sur un langage existant, elles sont relativement accessibles et permettent l'intégration de codes pré-existants. Une variante de cette technique consiste à intégrer ses mécanismes directement au sein d'un compilateur existant comme `gcc` pour le rendre capable de reconnaître des schémas spécifiques comme des séries d'instructions ALTIVEC ou des structures de communications MPI. Cette extension permet alors au compilateur hôte de générer un code en totale adéquation avec la classe de problème considérée. Elle assure ainsi un contrôle total sur l'efficacité de l'application finale. Néanmoins, l'inclusion de telles optimisations au sein d'un compilateur C ou C++, par exemple, est un travail complexe. En outre, une telle extension est potentiellement dépendante de la version du compilateur considérée voire de la plate-forme de développement, ce qui est relativement prohibitif. En outre, l'ajout d'éléments de syntaxe propres au parallélisme revient souvent à modifier en profondeur la syntaxe du langage hôte [74, 127], rapprochant ainsi l'utilisation de ces compilateurs ou générateurs de code de celle de nouveaux langages.

3.3.3 Approches basées sur des bibliothèques

Ces approches offrent un bon niveau d'expressivité et la variété des langages cibles utilisés permet d'envisager une bonne accessibilité et une intégration aisée avec du code existant grâce aux progrès des langages comme le C, le C++ ou le JAVA. La généralisation des approches telles que la programmation orientée objet permet ainsi la création de bibliothèques dont le niveau d'abstraction est très élevé. Il est en outre possible de développer ce type d'outils pour un grand nombre de langages cibles, permettant l'intégration aisée de code existant ou de bibliothèques tierces. Néanmoins, le code généré par ces bibliothèques est sou-

vent naïf et ne prend pas en compte des problématiques d’implantation liées au langage cible. Par exemple, la gestion de tableaux numériques par une approche objet en C++ est globalement 3 à 20 fois plus lente que sa version bas-niveau écrite en C [185] et l’utilisation de hiérarchie de classes dans des outils comme MUESLI [119] conduit à des pertes de performances allant de 20% à 110%. Ces contre-performances ne sont pas dues à une mauvaise conception de la part des développeurs mais proviennent souvent de la manière inefficace dont le langage constraint l’implémentation de ces abstractions.

3.3.4 Solution proposée

Nous pensons qu’au sein de la communauté Vision, l’utilisation de nouveaux langages ou compilateurs est à écarter. Un large panel de code existant est disponible dans de nombreux langages comme le C, le C++ ou MATLAB®. Ces fragments de code sont autant d’éléments réutilisables qu’il est inconcevable d’ignorer. En outre, l’aspect orienté objet du C++ offre un avantage non négligeable en terme d’expressivité. Il permet en effet de répondre au besoin d’abstraction nécessaire dans le développement de bibliothèques dédiées. Malheureusement, l’utilisation du C++ comme langage cible peut induire un manque d’efficacité. En effet, plusieurs phénomènes lié au modèle de compilation du C++ et au paradigme objet viennent réduire les performances des bibliothèques orientées objet.

Nous avons donc choisi de créer deux bibliothèques orientée objet en C++ dédiées respectivement à la programmation SIMD via un modèle de programmation à base de tableaux numériques et à la programmation MIMD à base de squelettes algorithmiques et évitant les écueils décrits précédemment.

La description de l’interface et de l’implantation de ces bibliothèques fait l’objet des deux chapitre suivants. Nous verrons comment chacune d’elle utilise un mécanisme d’implantation limitant les pertes de performances inhérentes à l’application du paradigme orienté objet en C++ afin de garantir leur efficacité.

Chapitre 4

La bibliothèque E.V.E.

«Until Eve arrived, this was a man's world.»

Richard Armour

La difficulté principale dans la conception d'outils haut niveau pour la programmation SIMD basée sur l'extension ALTIVec est de proposer un schéma d'implantation qui allie abstraction et performances. Ceci explique que de tels outils restent peu développés. Nous nous proposons de combler ce manque en développant une bibliothèque orientée objet permettant de tirer parti du parallélisme fourni par l'unité ALTIVec de manière efficace tout en conservant un haut niveau d'abstraction : E.V.E.¹ [70, 71].

Au sein de ce chapitre, nous évoquons successivement le modèle de programmation de E.V.E. et les limitations de l'implantation classique de ce dernier au sein d'un langage objet. Nous présentons ensuite l'interface effective fournie par E.V.E. puis nous étudions en détails les solutions techniques qui nous permettent de nous abstraire des limitations mise en avant. Enfin, nous évaluons les performances de E.V.E. à travers une série de tests simples et concluons par une étude de cas plus complexe.

4.1 Modèle de programmation

La définition d'une bibliothèque permettant d'utiliser au mieux les possibilités fournies par ALTIVec nécessite de considérer plusieurs points :

¹*Expressive Velocity Engine*

- **Quelles limitations l'interface d'ALTIVEC impose-t-elle sur E.V.E. ?**

Les limitations de l'extension ALTIVEC exposées dans la section 2.6.3 influent grandement sur la structure d'une bibliothèque de calcul scientifique. L'absence de support pour les réels double précision et la complexité des fonctions de transtypage nous forcent à ne vectoriser que les expressions homogènes contenant des types compatibles avec ALTIVEC. Ainsi, une expression utilisant des tableaux de types différents ne pourra être vectorisée. En outre, l'asymétrie du jeu d'instructions limite le nombre d'opérations vectorisables et nécessite de développer des fonctions adéquates pour combler les manques les plus flagrants.

- **Quel schéma de vectorisation utiliser ?**

Les aspects architecturaux de l'extension ALTIVEC demandent de prendre en compte des paramètres comme le remplissage du cache et du pipeline. Ces paramètres de relativement bas niveau doivent se fondre de manière naturelle au sein de l'interface pour ne pas compliquer la tâche du développeur et rendre le code inutilement verbeux. Pour remplir correctement le cache de données et les divers pipelines de l'unité ALTIVEC, par exemple, il est nécessaire de parcourir les données contenues dans les tableaux numériques non pas en chargeant les vecteurs un par un mais par blocs contigus en effectuant un déroulage – partiel ou total – de la boucle de traitement. Il va donc falloir permettre à l'utilisateur final de maîtriser ce déroulage.

- **Comment gérer les schémas de vectorisation irréguliers ?**

Il existe un certain nombre d'algorithmes qui ne peuvent être exprimés vectoriellement de manière régulière. Ainsi les opérations de tri, de recherche ou de filtrage numérique [52, 69] nécessitent de mettre en place un schéma irrégulier de chargement ou de calcul. La plupart de ces algorithmes se décomposent en trois phases : une première phase dans laquelle les données sont pré-traitées afin de modifier leur placement dans la mémoire; une deuxième phase où un algorithme vectoriel régulier est appliqué sur ces données et enfin, une dernière phase dans laquelle les résultats sont modifiés afin de se conformer à la structure mémorielle des données initiales. D'autres algorithmes irréguliers utilisent une approche récursive du problème ou un schéma d'accès non-contigu. Comme ce modèle d'évaluation diffère fortement de celles fonctions régulières, il est nécessaire de distinguer ces évaluations des évaluations régulières et de proposer une mécanisme qui permet de les différencier et de proposer pour chacune un mode de calcul adéquat.

La solution proposée par E.V.E. est de fournir un modèle à base de **marqueurs locaux** qui permettent de définir les points précis d'un algorithme qui devront être optimisés et de quelle manière : déroulage, vectorisation via ALTIVEC ou prévention des copies redondantes, etc. Ceci nous permet alors de définir un modèle de programmation explicite — à la manière de HPF [78, 115] par exemple — dans lequel ces marqueurs permettent d'expliciter la nature vectorielle des **données** et non des opérations, permettant alors de décrire des algorithmes ou le vectoriel et le séquentiel peuvent se mêler. Une interface unique est alors définissable et sera capable, dans chacune de ces situations, de fournir des performances élevées.

De manière assez globale, cette interface doit permettre de manipuler des tableaux comme une seule entité et d'y appliquer toute sorte de fonctions arithmétiques, logiques ou algébriques en utilisant une syntaxe simple. Cette hypothétique interface permettrait donc d'écrire un code semblable à celui du listing 4.1.

Listing 4.1 – Interface type d'une classe de gestion des tableaux numériques

```

1 array<float> a(N), b(N), c(N), d(N);
2
3 a = b+c+d;
```

L'utilisation des optimisations SIMD se déclencherait donc via une écriture proche de celle exposée dans le listing 4.2.

Listing 4.2 – Interface d'une classe de gestion des tableaux numériques SIMD

```

1 array<float, simd> a(N), b(N), c(N), d(N);
2
3 a = b+c+d;
```

La simple **présence** du marquer `simd` dans la déclaration des tableaux `a`, `b` et `c` va entraîner la vectorisation — via l'unité SIMD sous-jacente (ici ALTIVEC) — du code produit. Lorsque des données marquées comme SIMD et des données non-marquées sont utilisées au sein d'un même calcul, E.V.E. sera capable d'évaluer l'ensemble de l'expression en effectuant un calcul purement séquentiel. Il est alors de la responsabilité du développeur de limiter ces calculs «mixtes» afin de conserver des performances élevées.

4.1.1 Problématique de l'implantation orientée objet

L'implantation d'une telle interface pose des problèmes classiques en C++. Ainsi, il est possible de surcharger un ensemble d'opérateurs de base pour simplifier l'écriture de certaines méthodes ou fonctions. Lorsque le compilateur traite une série d'appel d'opérateurs surchargés, il procède à une évaluation successive de chaque opérande et génère un objet temporaire. Le listing 4.3 présente un exemple de surcharge des opérateurs + et = pour une classe de tableaux numériques. Dans le cas classique, l'opérateur + crée un tableau temporaire contenant le résultat de l'opération et le renvoie sur la pile.

Listing 4.3 – Surcharge de l'opérateur +

```

1 array<T> operator+(const array<T>& l, const array<T>& r)
2 {
3     array<T> r(l.size());
4     for(size_t i=0;i<res.size();i++) r[i] = a[i] + b[i];
5     return r;
6 }
```

L'opérateur = qui effectue l'affectation d'un tableau à un autre se définit comme sur le listing 4.4

Listing 4.4 – Surcharge de l'opérateur d'affectation

```

1 array<T>& array<T>::operator=(const array<T>& src)
2 {
3     resize(src.size());
4     for(size_t i=0;i<size();i++) res[i] = src[i];
5     return *this;
6 }
```

Lorsque l'on compose les appels à ces opérateurs, le compilateur les résout de manière dyadique, créant autant de tableaux temporaires que nécessaire. Plus précisément, le listing 4.1 génère un code intermédiaire présenté sur le listing 4.5. Pour des tableaux de petite taille, la perte de performance provient principalement du surcoût dû aux allocations et libérations de la mémoire. Pour des tableaux de taille moyenne², le surcoût provient majoritairement des accès mémoires supplémentaires qui provoquent des sorties de cache prématuées.

²On considère une zone mémoire comme étant de taille moyenne si cette taille est de l'ordre

Listing 4.5 – Résolution dyadique de la surcharge d’opérateurs

```

1 double* _t1 = new double[N];
2 for (int i=0; i < N; ++i) _t1[i] = b[i] + c[i];
3 double* _t2 = new double[N];
4 for (int i=0; i < N; ++i) _t2[i] = _t1[i] + d[i];
5 for (int i=0; i < N; ++i) a[i] = _t2[i];
6 delete [] _t2;
7 delete [] _t1;

```

Pour les tableaux de grande taille, le surcoût provient du calcul et de l'accès aux variables temporaires. Si l'on considère le code C optimal équivalent (listing 4.6)

Listing 4.6 – Code C optimal pour l’addition de tableaux numériques

```

1 for (int i=0; i < N; ++i) a[i] = b[i] + c[i] + d[i];

```

on montre que en moyenne, l'évaluation d'une expression de M opérandes et N opérateurs s'exécutent $\frac{2N}{M}$ fois moins vite que le code équivalent en C ou en FORTRAN [185].

On pourrait croire que des phases d'optimisation intensive peuvent éliminer ces problèmes. En général, il n'en est rien car les compilateurs n'ont pas accès à la sémantique des abstractions mises en oeuvres. Là où un développeur voit un ensemble d'opérations sur des tableaux numériques, le compilateur ne voit que des boucles et des allocations mémoires. La solution d'inclure des notions de sémantiques au sein de bibliothèques a été envisagée mais aucun de ces travaux n'a réellement abouti car chaque bibliothèque utilise des couches d'abstractions différentes et nécessite des stratégies d'optimisation spécifiques.

Une approche plus efficace est de développer des bibliothèques qui proposent à la fois un niveau d'abstraction élevé et un mécanisme d'optimisation *ad hoc*. Ce concept de «bibliothèque active» [191] met en oeuvre une phase d'optimisation haut-niveau dépendant du modèle de programmation, laissant le compilateur effectuer les optimisations bas-niveau usuelles (allocation des registres et ordonnancement des instructions par exemple). Pour réaliser de telles librairies, plusieurs systèmes de métadéfinition ont été proposés. Parmi ceux ci, on peut citer

de grandeur de celle du cache L1 du processeur

des projets comme Xroma [50], MPC++ [102], Open C++ [38] ou Magik [63]. Ces systèmes permettent d’injecter au sein du compilateur C++ des extensions qui permettent de fournir à ce dernier les indications sémantiques nécessaires à une compilation efficace. L’utilisation de ces outils reste néanmoins difficile et souffre des mêmes limitations que les approches basées sur l’écriture de nouveaux compilateurs. Il est donc nécessaire de chercher une alternative intégrable directement au sein du langage C++ sans nécessiter d’outils externes. Nous verrons au paragraphe 4.3 comment un tel mécanisme peut s’appliquer à l’implantation de l’interface de E.V.E..

4.2 Interface utilisateur

En intégrant cette technique aux éléments usuels d’une classe de tableaux numériques, E.V.E. fournit une interface dont les performances s’affranchissent des problèmes de performances évoqués précédemment. Ce paragraphe va exposer les différents éléments de l’interface de E.V.E. : ses classes de conteneurs, les marqueurs d’optimisations disponibles et les fonctions manipulant ses conteneurs.

4.2.1 Les classes `array` et `view`

Comme nous l’avons dit précédemment, le but de E.V.E. est de fournir une interface intuitive qui permet aux développeurs de la communauté Vision de prendre rapidement en main de tels outils. Le modèle de programmation à base de tableaux est implanté via les classes `array` et `view` qui font office de conteneurs génériques compatibles avec l’ensemble des types arithmétiques du C++. La principale différence entre `array` et `view` est la manière dont ces classes se comportent vis à vis de la gestion de la mémoire. `array` est capable d’allouer et de libérer la mémoire nécessaire au stockage de ces éléments de manière transparente. `view` permet quant à elle d’utiliser des zones mémoires provenant d’autres sources – une autre bibliothèque ou un système d’acquisition vidéo par exemple – sans provoquer de copie et fait office d’adaptateur [84] entre les classes et fonctions de E.V.E. et des codes tiers. Les classes `array` et `view` sont définies ainsi :

```
template<typename T, typename O> class array;
template<typename T, typename O> class view;
```

Les paramètres *templates* de `array` sont `T` qui définit le type d’élément contenu dans le tableau et `O` qui définit les différents marqueurs d’optimisations.

4.2.2 Choix des options d'optimisations

Ces marqueurs vont permettre lors de l'instanciation d'une array ou une view de générer un code adapté à un grand nombre de situations :

- **L'activation des optimisations SIMD.**

De nombreux facteurs comme le type de données et le type de fonctions utilisés vont modifier le type d'optimisation applicable au moment de l'évaluation des expressions mettant en jeu des instances de array ou view. L'utilisation du marqueur `simd` va indiquer au noyau d'optimisation *template* que les expressions utilisant ces instances de array ou view pourront utiliser les optimisations à base de primitives ALTIVec.

- **L'activation d'un déroulage des calculs.**

Lors de l'évaluation d'une expression, il est possible de forcer E.V.E. à dérouler la boucle d'évaluation d'expressions d'un facteur défini par l'utilisateur. Ce déroulage permet dans certains cas de remplir de manière optimale le cache de données et d'augmenter les performances du calcul en cours.

- **Une stratégie d'allocation mémoire.**

L'allocation de la mémoire d'un array peut être dynamique, statique ou être déléguée à une classe définie par l'utilisateur. Par défaut, array alloue sa mémoire de manière dynamique. Évidemment, cette option n'a aucun effet sur les view car ces dernières ne font office que d'adaptateur entre E.V.E. et des zones mémoires externes.

- **L'indexation de la matrice.**

Chaque array ou view peut définir sa base d'indexation. Par défaut, l'indexation des éléments d'une instance de array ou view commence à 0 comme en C.

Le listing 4.7 donne un exemple de déclaration de array et de view utilisant diverses combinaisons de marqueurs. Ces marqueurs sont passés en arguments aux classes array ou view via la métा-fonction `settings` qui s'occupe d'éliminer les doublons de la liste, de l'ordonner et d'émettre d'éventuels messages d'erreur si des paramètres sont utilisés avec des valeurs hors-limites (comme un pas de déroulage négatif par exemple). On notera que l'ordre des marqueurs n'est pas imposé. En outre, si des options incompatibles avec le type de conteneur sont spécifiées, un message d'erreur sera émis lors de la compilation. La puissance de

cette approche à base de marqueurs réside dans le grand nombre de cas particuliers que l'on peut ainsi gérer sans avoir eu à définir autant de classes différentes que de cas particuliers.

Listing 4.7 – Instanciation des classes `array` et `view`

```

1 // Tableau standard :
2 array<int> a;
3
4 // Tableau optimise Altivec :
5 array<double, settings<simd> > d;
6
7 // Vue FORTRAN de d:
8 view<complex<float>, settings<base_index<1> > > b(d);
9
10 // Tableau 4x4 allouant statiquement 20 octets:
11 array<char, settings<static_storage<20> > > c(ofSize(4, 4)
12     );
13
14 // Vue utilisant un pas de déroulage de 2,
15 // et les optimisations Altivec
16 view<float, settings<unroll<2>, simd > > e;

```

Afin de simplifier le développement d'applications et de ne pas noyer l'utilisateur final sous une multitude de déclarations utilisant des listes de marqueurs diverses et variées, E.V.E. fournit un fonction `view_as<S>` dont le paramètre *template* permet de spécifier à la volée de nouveaux marqueurs sur des instances existantes de `array` ou `view` sans effectuer de recopie (listing 4.8).

Listing 4.8 – Utilisation de `view_as`

```

1 array<int> a;
2
3 view_as< settings<simd> >(a) = rand(4, 4);
4 view_as< settings<unroll<2> >(a) = 2*a;

```

4.2.3 Fonctions disponibles

E.V.E. fournit ensuite un ensemble de fonctions et de surcharges d'opérateurs applicables sur `array` et `view`. Ces fonctions peuvent être classées en sept grandes

classes : des opérations arithmétiques et booléennes, des fonctions de remplissage, des fonctions modifiant la forme des tableaux, des fonctions issues de `math.h`, des fonctions d’interfacage avec les bibliothèques BLAS et LAPACK, des fonctions de réduction et des fonctions de traitement du signal. Dans tout les cas, leurs prototypes et leur comportement se calquent de manière fidèle sur ceux des fonctions MATLAB® équivalentes, si elles existent, dans l’optique de faciliter la réutilisation de code MATLAB® qui sont très utilisés en vision artificielle pour le prototypage d’applications.

4.2.3.1 Opérateurs arithmétiques et booléens

Ces opérateurs génèrent un conteneur dont les éléments sont le résultat de l’application de leur équivalent C++ à chaque élément du ou des conteneurs d’entrées. La totalité des opérateurs classiques sont ainsi surchargés, exception faite de l’opérateur ternaire ? : qui, n’étant pas surchargeable, est remplacé par la fonction `where`. La quasi-totalité de ces fonctions sont vectorisables aux exceptions près de certaines fonctions spécifiques aux nombres complexes et aux réels double précision.

Listing 4.9 – Exemple d’opérations arithmétiques classiques

```

1 // r[i] = a[i] < b[i] ? a[i]+b[i] : a[i]*b[i]
2 r = where(a < b, a+b, mul(a,b));

```

4.2.3.2 Fonctions issues de `math.h`

Ces fonctions effectuent les mêmes opérations que leurs équivalents issus de l’en-tête standard `math.h`, comme les fonctions trigonométriques ou les fonctions d’arrondi. Des versions spécifiques sont fournies pour les types de données qui ne sont pas naturellement supportées par le standard – comme par exemple le calcul du module ou de l’argument d’un complexe ou le cosinus d’un entier.

Listing 4.10 – Interface avec `math.h`

```

1 //b[i] = 2*cos(a[i]) + sqrt(a[i]);
2 b = 2*cos(a) + sqrt(a);

```

4.2.3.3 Fonctions de réduction

Ces fonctions effectuent des calculs dont le résultat est un tableau possédant moins de dimensions que le ou les tableaux d'entrées. Ces fonctions sont par exemple la somme d'un tableau, l'évaluation d'un prédictat ou le calcul du maximum d'un tableau

Listing 4.11 – Evaluation du prédictat `any`

```

1 // k est vrai si il existe i tel que : a[i] > sum(b)
2 bool k = any( a > sum(b) );

```

4.2.3.4 Fonctions de remplissage

Ces fonctions permettent de remplir des conteneurs avec des valeurs numériques constantes ou définies par diverses formules. Ces fonctions permettent en outre de fournir un remplacement pour la syntaxe de création de tableau fourni par MATLAB® comme l'opérateur «`:`» où le listage explicite des éléments d'une matrice.

Listing 4.12 – Fonction de remplissage et leur équivalent MATLAB®

```

1 a = ones(1,4);           // a = [1 1 1 1]
2 b = iota(1,0.3,2);     // b = [1:0.3:2]
3 c = cons(2,4,6,8);     // c = [2 4 6 8]

```

4.2.3.5 Fonctions de modification de forme

Ces fonctions permettent de modifier la forme d'un tableau par concaténation ou extraction de sous-tableaux. Des classes spécifiques – All et Range – remplacent en outre la syntaxe d'indexation multiple de MATLAB®.

Listing 4.13 – Fonction de modification de forme et leur équivalent MATLAB®

```

1 a = cons(1,2,3,4);           // a = [ 1 2 3 4 ]
2 b = catv(a,a);             // b = [ a;a ]
3 c = b(Range(Begin(),End())); // c = b(1:end)
4 d = b(All(),Range(0,2,3)); // d = b(:,1:2:4)

```

4.2.3.6 Fonctions BLAS et LAPACK

E.V.E. fournit une interface haut-niveau vers un sous-ensemble des fonctions fournies par BLAS et LAPACK. Ces fonctions sont des implantations très efficaces de routines d'algèbre linéaire comme par exemple le produit matriciel, la résolution de systèmes linéaires ou la décomposition SVD. BLAS et LAPACK sont, pour une plate-forme donnée, implantés de manière à tirer partie de l'architecture de cette plate-forme. Ceci implique que les fonctions LAPACK ou BLAS utilisée par E.V.E. sont vectorisées par défaut, quel que soit le marquage des tableaux E.V.E..

Listing 4.14 – Interface avec LAPACK

```

1 // Resolution d'un systeme AX = B
2 // en MATLAB : x = a \ b
3 mldivide(a,x,b);
4
5 // Verification
6 cout << globalMax(abs(a*x-b)) << endl;

```

4.2.3.7 Fonctions de traitement du signal

Ces fonctions permettent de gérer naturellement l'application de filtres numériques aux éléments d'un array ou d'un view. Ces filtres peuvent être composés de manière naturelle afin de simplifier leur définition par l'intermédiaire de la classe filter

Listing 4.15 – Applications d'un filtre gaussien

```

1 filter<char, settings<simd> > gauss_x = cons(1,2,1);
2 r = (gauss*gauss)(a);

```

L'ensemble des ces fonctions permettent à E.V.E. d'exprimer un large panel d'algorithmes de calculs scientifiques et de traitement d'images. Nous verrons au paragraphe 4.4 et au chapitre 6 une portion de l'expressivité apportée par cette interface.

4.3 Implantation

Une des techniques permettant de s’abstraire des limitations évoquées au paragraphe 4.1.1 consiste à analyser le code source initial et à utiliser un mécanisme dit d'**évaluation partielle** [106]. Effectuer l’évaluation partielle d’un code source consiste à discerner les parties statiques du programme – c’est à dire les parties du code calculables à la compilation – des parties dynamiques – calculables à l’exécution. Le compilateur est alors amené à évaluer cette partie statique et à générer un code ne contenant plus que les parties dynamiques. Dans le cadre du calcul scientifique, une de ces techniques est classiquement employé afin de limiter l’impact de la réduction dyadiques des opérateurs.

En sus de ces techniques d’évaluation partielle, plusieurs aspects de l’implantation de E.V.E. mettent en jeu des techniques de métaprogrammations dédiées à la résolution de problèmes spécifiques aux méthodes de vectorisation. Il s’agit en particulier du calcul de la «vectorisabilité» d’une expression et de la détection de schémas vectoriels spécifiques.

4.3.1 L’évaluation partielle en C++

Afin de déterminer les portions de code à évaluer statiquement, un évaluateur partiel doit analyser le code source original et étiqueter les structures et les données. Cet étiquetage permet par la suite de spécialiser les fragments de code. Ceci nécessite un langage à deux niveaux dans lequel il est possible de manipuler de tels marqueurs.

Un tel langage est accessible au sein du C++ via l’utilisation du mécanisme des *templates*.³ En effet, les *templates* C++ se comportent comme un tel langage à deux niveaux. Les fonctions et classes *templates* utilisent à la fois des arguments statiques – les paramètres *templates* – et dynamiques – les paramètres de la fonction en elle-même. Prenons par exemple le prototype de la fonction `sum` destinée à sommer les éléments d’un tableau numérique :

Listing 4.16 – Prototype de la fonction *template* `sum`

```

1 template<int N, typename T>
2 inline T sum( T* array );

```

³Un rappel sur les principales caractéristiques des *templates* est donné dans l’annexe I.

Dans le listing 4.16, `N` et `T` sont des données statiques qui apparaissent comme des paramètres *templates* alors que `array` est une donnée dynamique qui apparaît comme un argument de fonction classique. La compilation d'une instance de `sum<N>(T*)` nécessite d'évaluer ses paramètres statiques (`N` et `T`) avant de pouvoir effectuer l'évaluation de ses paramètres dynamiques. Ce point de vue nous permet alors de considérer les types C++ comme étant des données à part entière au sein d'un langage spécifique capable de les manipuler. On peut alors se demander si ce deuxième niveau de langage fourni par les *templates* est suffisamment expressif.

On démontre par construction [187] que les *templates* C++ forment un langage Turing-complet, c'est à dire qu'ils permettent de représenter toutes les fonctions calculables au sens de Turing. Cette propriété permet de montrer que, moyennant une réécriture potentiellement non-triviale, tout programme peut être exprimable sous la forme d'un programme utilisant les *templates* C++. Un des premiers exemples de tels programmes fut présenté par Erwin Unruh[181]. Ce programme ne s'exécutait pas mais renvoyait une série de messages à la compilation qui énumérait la liste des `N` premiers nombres premiers (cf. Annexe II). Il démontrait ainsi que les *templates* permettaient d'exprimer plus que la simple générnicité des classes et des fonctions.

Plusieurs techniques de base ont donc été mises au point [191] pour développer une large variété de programmes statiques – ou méta-programmes – à même de faciliter la manipulation de ce «langage dans le langage» que constituent les *templates*. D'une manière générale, ces techniques de méta-programmation consistent à utiliser l'instanciation des *templates* C++ pour «exécuter» un programme à la compilation. Cette «exécution» passe par l'évaluation, la mise en correspondance de valeurs constantes ou de types. Une fois l'évaluation de ces méta-programmes effectué, le compilateur génère un code C++ sans aucun *template* et prêt à être compilé normalement. Ce mécanisme se comporte alors comme un évaluateur partiel car on y retrouve les grandes étapes de marquage des données statiques – via la spécification d'argument de type – et de génération de code.

Un exemple d'une telle évaluation partielle est donnée dans le listing 4.3.1. La fonction `volumeOfCube` effectue le calcul du volume d'un cube d'arête `length` en utilisant une fonction `pow(x, N)` qui calcule x à la puissance N . Dans ce listing, les parties statiques ont été sur-lignées et on voit nettement que l'utilisation de `pow` n'est pas la solution optimale. En effet, le temps de traitement de la boucle `for` représente la large partie du temps de calcul pour de petite valeurs de N .

```

1  float volumeOfCube(float l) { return pow(l, 3); }

2

3  float pow(float x, int N)
4  {
5      float y = 1;
6      for(int i=0; i < N; i++) y *= x;
7      return y;
8  }

```

Grâce aux *templates*, on peut définir un ensemble de méta-programmes qui explicite cette nature (listing 4.3.1).

```

1  template<size_t I, size_t N> struct meta
2  {
3      static inline float Pow( float x )
4      {
5          return x*meta<I++,N>::Pow(x);
6      }
7  };
8
9  template<size_t N> struct meta<N,N>
10 {
11     static inline float Pow( float x ) { return 1; }
12 };
13
14 template<size_t N> float pow(float x)
15 {
16     return meta<0,N>::Pow(x);
17 }
18
19 float volumeOfCube(float l) { return pow<3>(l); }

```

Dans cette version, plusieurs éléments spécifiques à la manipulation des *templates* sont utilisés :

- La fonction `pow` utilise explicitement un paramètre statique — le paramètre *template* `N` — en lieu et place de son paramètre dynamique.
- Une métaprogramme remplace la boucle `for` principale. Cette structure uti-

lise une forme récursive et une spécialisation partielle afin d'effectuer l'ensemble des calculs nécessaires. On remarque la version statique de l'incrément de l'index de boucle au sein de la méthode statique `meta<I, N>::Pow`.

Lors de l'appel de `volumeOfCube`, le compilateur va évaluer les divers appels *templates* et générer un code en lieu et place de l'appel à `pow<3>`. Le code final est alors le suivant :

```

1 float volumeOfCube(float l)
2 {
3     return l*l*l*l;
4 }
```

Le point important à noter est que **l'ensemble de ces calculs et génération de code sont effectués au moment de la compilation**. Aucun appel de fonction n'est généré et les performances de cette version statique de `volumeOfCube` sont supérieures à la version entièrement dynamique [190]. Il existe un grand nombre de techniques identiques permettant d'exprimer la quasi-totalité des structures de contrôles et des constructions disponibles en C et en C++.

4.3.2 Les *Expression Templates*

Parmi ces méthodes d'évaluation partielle à base de *templates*, les *Expression Templates* sont une technique dont le but est de créer un noyau d'analyse syntaxique et de génération de code au sein d'un code source C++ [188, 93, 112]. Ce générateur de code statique permet alors de procéder à l'évaluation partielle, au moment de la compilation, d'arbres de syntaxe abstraite.

L'utilisation principale de cette technique est l'optimisation de l'évaluation d'expressions arithmétiques mettant en oeuvre des opérandes dont les types sont définis par l'utilisateur. Comme nous l'avons vu précédemment, la génération du code correspondant à des expressions utilisant des surcharges d'opérateurs est peu efficace. Prenons par exemple le cas d'une classe de tableaux numériques définie de manière classique (cf listing 4.17). Cette forme classique est caractérisée par le fait que l'opérateur d'affectation défini à la ligne 7 prend en argument une instance de `array<T>`. Au sein de cette méthode, le tableau courant est d'abord redimensionné. Ensuite, le contenu du tableau passé en argument est copié élément par

élément dans la zone mémoire du tableau courant. L'idée maîtresse des *Expression Templates* est de modifier la sémantique de ces opérateurs afin d'optimiser l'affectation finale de leur évaluation.

Listing 4.17 – Classe de tableau numérique classique

```

1  template<typename T> class array
2  {
3      public:
4      // ...
5      array<T>& operator=( const array<T>& src )
6      {
7          resize(src.mSize);
8          for(size_t i=0; i<mSize; i++) mData[i] = src.mData[i];
9          return *this;
10     }
11
12     void resize( size_t sz );
13
14     private:
15     T* mData;
16     size_t mSize
17 };
```

Considérons une expression mettant en jeu des opérateurs manipulant un classe de tableaux numériques :

$$r = a + b + c$$

Comme nous l'avons vu dans la section 4.1.1, cette expression s'évalue en trois passes : évaluation de $a + b$, de la somme de ce résultat intermédiaire avec c et finalement la recopie du résultat final dans r .

$$\begin{aligned} \forall i, t_1[i] &= a[i] + b[i] \\ \forall i, t_2[i] &= t_1[i] + c[i] \\ \forall i, r[i] &= t_2[i] \end{aligned}$$

Le but est alors de se ramener à une évaluation **en une seule passe** :

$$\forall i, r[i] = a[i] + b[i] + c[i]$$

Pour cela, il faut s'intéresser à la **structure** de l'expression à droite de l'affectation. La structure de l'arbre de syntaxe abstraite associé à cette expression est

clairement fixée au moment de l'écriture de cette expression, alors que le contenu des tableaux n'est connu qu'à l'exécution. Cette dichotomie montre que, dans l'évaluation d'une telle expression, la composition des opérateurs est une donnée statique et que les données contenues dans les tableaux sont dynamiques. L'application d'un évaluateur partiel à une expression comprenant M opérandes et N opérateurs revient à créer lors de la compilation une fonction φ à M arguments effectuant l'application des N opérateurs de l'expression initiale sur une unique valeur scalaire. A l'exécution, seule l'application de cette fonction composite aux données initiales sera évaluée. Ici, nous obtenons :

$$\varphi(x, y, z) = x + y + z$$

$$\forall i, r[i] = \varphi(a[i], b[i], c[i])$$

L'idée est donc de définir un certain nombre de classes permettant de représenter **au sein du langage** la structure même de l'arbre de syntaxe abstraite afin de pouvoir «construire» φ au moment de la compilation. Pour cela, nous définissons plusieurs classes *templates* qui vont nous permettre de représenter respectivement la structure de l'arbre, ses éléments terminaux et les opérations effectuées à la traversée des nœuds de l'arbre. L'ensemble de ces classes va alors évaluer partiellement l'expression à la compilation et générer un code résiduel correspondant à la suite d'opérations strictement nécessaires pour une évaluation dynamique optimisée.

4.3.2.1 Topologie de l'arbre de syntaxe abstraite.

Au lieu de chercher à évaluer une instance de `array<T>` et de l'affecter à une autre instance de `array<T>`, nous allons revenir à la définition purement grammaticale d'expressions arithmétiques mettant en jeu des tableaux numériques. On peut ainsi définir un ensemble réduit de règles pouvant décrire une expression arithmétique quelconque constituée des quatre opérateurs arithmétiques de bases (+, -, *, /) et dont les éléments terminaux sont soit des tableaux numériques (`array`), soit des constantes numériques (`constant`).

```

val      := array | constant
OP       := + | - | * | /
node     := expr OP expr
expr     := val | node

```

Afin de pouvoir manipuler la structure de l’arbre de syntaxe abstraite d’une expression construite à partir de cette grammaire, nous allons l’encoder au sein d’une classe *template*. Pour ce faire, nous utilisons la nature intrinsèquement récursive de l’arbre et des *templates*. Cette structure récursive est donnée par la règle de grammaire *expr* := *val* | *node* qui définit une **expression** comme étant soit une valeur ou soit un élément de type noeud. Nous introduisons donc la classe *expression* (listing 4.18) qui permet de manipuler cet élément de la grammaire.

Listing 4.18 – Élément de base d’un arbre de syntaxe abstraite *template*

```

1 template<class E> class expression
2 {
3     public :
4     typedef typename E::return_t return_t;
5
6     expression( const E& xpr) : mExpr(xpr) {}
7     return_t load_val( size_t i ) const
8     {
9         return mExpr.load_val(i);
10    }
11
12 private:
13     E mExpr;
14 }
```

La classe *expression* présente plusieurs points importants :

- Son paramètre *template* *E* représente le contenu de l’expression. Le type de *E* est quelconque. En particulier, *E* peut être une instance du *template* *expression* lui-même. Ce comportement récursif nous permet, comme nous le détaillerons plus loin, de former, par composition, une expression de complexité arbitraire.
- La méthode *load_val()* de la classe *expression* est le point d’entrée de l’évaluateur partiel. Lorsque ce dernier est appelé, il déclenche l’évaluation récursive de l’instance *mExpr* de *E*.
- Le type de retour de la méthode *load_val()* est défini de manière récursive. La directive *typedef* de la ligne 3 accède au type *return_t* défini au sein de *E*.

Il faut maintenant définir le contenu d'une expression. Pour cela, nous nous intéressons dans un premier temps aux éléments de type nœud. Dans notre exemple simplifié, chaque nœud est un nœud binaire qui possède un fils droit, un fils gauche et un foncteur décrivant l'opération associée au nœud. Cette topologie est reprise dans la classe `binary_node` (listing 4.19) qui présente une structure similaire à celle de la classe `expression`. On retrouve ainsi :

- La méthode `load_val()` qui permet à l'évaluation partielle de se propager au sein des fils du nœud courant et d'insérer dans celle-ci le code effectif des opérateurs constituant l'expression initiale.
- Le type `return_t` qui permet à `binary_node` d'être inclus au sein d'une instance du type `expression`.

Listing 4.19 – Nœud d'un arbre de syntaxe abstraite *template*

```

1  template<class L, class R, template<class, class> class F>
2  class binary_node
3  {
4      public:
5          typedef F<L, R> func_t;
6          typedef typename func_t::return_t return_t;
7
8          binary_node(const L& l, const R& r) : mL(l), mR(r) {}
9          return_t load_val( size_t i ) const
10         {
11             return func_t::Load_Val(mL.load_val(i),
12                                     mR.load_val(i));
13         }
14
15     private:
16     L mL;
17     R mR;
18 }
```

Le point intéressant est ici la définition du foncteur décrivant l'opération à effectuer à la traversée du nœud. Le paramètre *template* `F` possède une signature bien particulière. Il s'agit d'un paramètre *template template*, c'est à dire qu'il

représente un argument statique qui accepte lui même des paramètres. L'utilisation de `F` nécessite bien sûr d'instancier ses paramètres *templates*. Pour ce faire, nous utilisons les types `L` et `R` contenant les informations de types des fils du noeud courant.

4.3.2.2 Données à évaluer à l'exécution.

La deuxième famille de classes va prendre en charge l'encodage statique des données de type `val`. Ces données représentent soit des tableaux d'éléments soit des constantes numériques. Nous allons dans un premier temps redéfinir une classe `array` afin de la rendre compatible avec la classe `expression`. Pour ce faire, nous ajoutons à `array` la définition du type `return_t` ainsi que la méthode `load_val`.

Listing 4.20 – Feuille de type tableau d'un arbre de syntaxe abstraite *template*

```

1 template<class T> class array
2 {
3     // ...
4     typedef T return_t;
5     return_t load_val(size_t i) const { return mData[i]; }
6 }
```

Nous définissons ensuite une classe `leaf` qui représentera les éléments de types valeurs numériques constantes.

Listing 4.21 – Elément constant d'un arbre de syntaxe abstraite

```

1 template<class T> class leaf
2 {
3     public:
4     typedef T return_t;
5
6     leaf( const return_t& val) : mVal(val) {}
7     return_t load_val( size_t ) const { return mVal; }
8
9     private:
10    return_t mVal;
11 }
```

De la même manière, leaf expose les types et méthodes nécessaires à son insertion au sein d'une expression.

4.3.2.3 Opérations aux nœuds de l'arbre syntaxique.

Reste à définir les foncteurs qui contiennent l'opération arithmétique à appliquer à la traversée d'un noeud. Une telle classe va fournir l'ensemble des évaluations statiques permettant de déterminer le type de retour du noeud et de générer le code correspondant à l'opération associée. Par exemple la classe Func_Add représente l'utilisation de l'opérateur + au sein d'une expression (cf listing 4.22):

Listing 4.22 – Exemple de foncteur d'un arbre de syntaxe abstraite *template*

```

1 template<class L, class R> struct Func_Add
2 {
3     typedef typename L::return_t           l_t;
4     typedef typename R::return_t           r_t;
5     typedef typename promotion<l_t,r_t>::type_t return_t;
6
7     static return_t Load_val(l_t l, r_t r) { return l+r; }
8 }
```

La structure de Func_Add suit le modèle classique déjà explicité. Nous retrouvons tout d'abord le type `return_t` qui permet de déterminer le type de retour du foncteur. `return_t` est calculé à partir des types de retour des fils droits et gauche du nœud binaire en utilisant une méta-fonction spécifique — `promotion` — qui effectue un calcul sur les types des opérandes et détermine le type strictement nécessaire au stockage du résultat de l'opération. Ensuite, la méthode de classe `Load_Val` est définie et contient le code strictement nécessaire à l'évaluation de la somme de deux valeurs scalaires. On peut définir de la même façon les classes `Func_Sub`, `Func_Mul`, `Func_Div`, etc.

4.3.2.4 Déroulement de l'évaluation partielle

Grâce à l'ensemble des classes ainsi définies, la construction des types correspondants à des expressions arithmétiques peut alors être effectuée. Le codage de l'expression $x + y + z$ via les classes Expr et Node se décompose en deux parties : le codage de $y + z$ et celui de $x + .$:

```
y + z : expression<binary_node<array<T>, array<T>, Func_Add> >
x + . : expression<binary_node<array<T>, . , Func_Add> >
```

Au final, nous obtenons un type très complexe dont la structure reflète celle de l’arbre de syntaxe abstraite initial :

```
expression< binary_node< array<T>,
            expression< binary_node<
                            leaf<T>,
                            array<T>,
                            Func_Add>
                        >
            >,
            Func_Add>
        >
    >
```

Il est évident que l’écriture explicite de tels types de donnée ne doit pas être laissée à la charge de l’utilisateur final. Pour cela, nous devons introduire un mécanisme qui permet à l’utilisateur d’écrire simplement des expressions et de générer le code statique correspondant. On utilise pour cela une simple surcharge des opérateurs arithmétiques du C++. Grâce à ces opérateurs, il est possible de décrire de façon simple et lisible des expressions tout en permettant leur encodage automatique sous la forme d’un type C++. Bien sur, il est nécessaire de fournir une version de chaque opérateur (+,-,*,/) pour chaque configuration possible des nœuds de l’arbre correspondant aux règles explicitées précédemment. Par exemple, l’opérateur + appliqué à des variables de types `array<T>` s’exprime maintenant comme présenté sur le listing 4.23.

Listing 4.23 – Surcharge de `operator+` générant une sous-expression template

```
1 template<class T>
2 expression<binary_node<array<T>, array<T>, Func_Add<T>>>
3 operator+(const array<T>& l, const array<T>& r)
4 {
5     typedef binary_node<array<T>, array<T>,
6                             Func_Add<T> > bn_t;
7     return expression<bn_t>( bn_t(l, r) );
8 }
```

À partir de ces mécanismes de construction, on obtient à la compilation une instance de la classe `expression` qui contient l'ensemble des informations de structure nécessaire à l'évaluation de l'arbre de syntaxe abstraite associé. Le code de l'opérateur d'affectation de `array<T>` est ensuite réécrit — et c'est là le point clé de cette méthode — afin d'utiliser le mécanisme d'évaluation partielle (listing 4.24).

Listing 4.24 – Surcharge de `operator=` évaluant une expression *template*

```

1 template<class T> template<class E>
2 array<T>& array<T>::operator=(const expression<E>& xpr)
3 {
4     for( int i=0; i<mSize; i++) mData[i] = xpr.load_val(i);
5     return *this;
6 }
```

L'évaluation proprement dite de l'expression `xpr` utilise la méthode `load_val` fournie par chacune des classes utilisées précédemment pour décrire les données statiques de l'expression. Lors de la compilation, l'appel de la méthode `load_val` de l'objet `xpr` va déclencher une série de résolutions d'appels récursifs de méthodes *templates*. Le code ainsi produit est déroulé automatiquement à l'emplacement précis où cette évaluation est demandée.

Si l'on reprend le code donné dans le listing 4.1, le code final ne contient donc aucun appel de fonction et aucune création de variable temporaire, ce que confirme la trace de l'évaluation partielle.

```

mData[i] = xpr.load_val(i)
mData[i] = expression<binary_node<...> >.load_val(i)
mData[i] = binary_node<array<T>, expression<...>, Func_Add>.load_val(i)
mData[i] = Func_Add.Load_Val(mL.load_val(i), mR.load_val(i))
mData[i] = mL.load_val(i) + mR.load_val(i)
mData[i] = a[i] + expression<binary_node< ... > >.load_val(i)
mData[i] = a[i] + binary_node<array<T>, array<T>, Func_Add>.load_val(i)
mData[i] = a[i] + Func_Add.Load_Val(mL.load_val(i), mR.load_val(i))
mData[i] = a[i] + mL.load_val(i) + mR.load_val(i)
mData[i] = a[i] + b[i] + c[i]
```

On peut pousser l'analyse jusqu'à examiner le code assembleur (tableau 4.1) pour juger plus finement de la pertinence de cette technique.

(a) <code>std::valarray<float></code>	(b) <code>array<float></code>	(c) <code>float[]</code>
<pre>L11: lwz r9,0(r3) slwi r2,r12,2 lwz r4,4(r3) addi r12,r12,1 lwz r11,4(r9) lwz r10,0(r9) lwz r7,4(r11) lwz r6,4(r10) lfsx f0,r7,r2 lfsx f1,r6,r2 lwz r0,4(r4) fadd f2,f1,f0 lfsx f3,r2,r0 fadd f1,f2,f3 stfd f1,0(r5) addi r5,r5,4 bdnz L11</pre>	<pre>L11: lwz r11,0(r4) lwz r5,24(r4) slwi r12,r8,2 lwz r2,48(r4) addi r8,r8,1 lfsx f4,r11,r12 lfsx f0,r5,r12 lfsx f3,r2,r12 fadds f2,f4,f0 fadds f1,f2,f3 stfsx f1,r12,r3 bdnz L11</pre>	<pre>L11: slwi r3,r2,2 addi r2,r2,1 lfsx f9,r3,r28 lfsx f10,r3,r29 lfsx f8,r3,r30 fadds f7,f9,f10 fadds f6,f7,f8 stfsx f6,r3,r27 bdnz L11</pre>

TAB. 4.1 – Comparaison des codes assembleurs finaux `valarray/array/C`

Le code généré ici correspond à l'addition de trois tableaux de réels simple précision : la colonne (a) en utilisant une implantation classique basée sur la classe de tableau numérique `valarray` fournie par la bibliothèque standard C++, la colonne (b) en utilisant le code présenté dans cette section et la colonne (c) en effectuant le calcul directement via une boucle écrite en C. On note que même si le compilateur est capable d'optimiser un grand nombre de boucles intermédiaires dans le cas de l'utilisation de `std::valarray`, le nombre d'accès mémoires (via les instructions `lfd`, `lwz` et `stfd`) et de calcul intermédiaires (via les instructions `fadd` et `addi`) reste bien supérieur à celui produit par l'utilisation de `array`. Le code produit par la technique d'*Expression Templates* présentée dans ce chapitre génère un code quasiment identique au code C, mis à part l'utilisation de primitives de chargement indirects. Cette qualité d'optimisation est due au fait que le mécanisme des *Expression Templates* constraint le générateur de code à suivre exactement le schéma voulu et ne fait quasiment pas d'appel aux routines d'optimisations du compilateur. En termes de performance, le code utilisant les *Expression Templates* bénéficie d'une vitesse d'exécution de l'ordre de 80 à 90 % de la vitesse d'exécution du code C équivalent [185] en fonction du type de donnée et de la taille des tableaux. En comparaison, le temps d'exécution du code utilisant

`std::valarray` ou une implantation équivalente n'atteint que 5 à 20 % de la vitesse d'exécution du code C équivalent. En contrepartie, le temps de compilation de tels programmes est sensiblement plus élevé — de l'ordre de deux à cinq fois plus lent — mais un tel surcoût est totalement contrebalancé par l'accélération notable obtenue à l'exécution et par le fait qu'il n'est payé qu'une fois lors de la compilation.

Les *Expression Templates* permettent donc de s'abstraire des problèmes de performances dus à la réduction dyadique des surcharges d'opérateurs. Dans le cadre du calcul scientifique, cette technique a fait ses preuves dans des bibliothèques telles que Blitz++ [189] ou POOMA [111]. Elle permet aussi d'enrichir la syntaxe de bibliothèques ayant une toute autre vocation. On peut citer par exemple la bibliothèque Spirit [56] qui permet d'utiliser une syntaxe proche de la forme normale de Backus-Naur [23] au sein de programme C++ et de définir des analyseurs syntaxiques à analyse descendante par une simple composition d'opérateurs.

4.3.3 Détermination de la «vectorisabilité» d'une expression

Pour être entièrement vectorisée, une expression mettant en jeu des instances des classes `array` ou `view` doit de respecter un certain nombre de contraintes :

- Les types de données contenus dans les tableaux doivent correspondre aux types supportés nativement par ALTIVEC. Des expressions mettant en jeu des réels double précision ou des nombres complexes par exemple ne pourront être vectorisées.
- L'ensemble de l'expression doit utiliser un unique type de donnée. La complexité des opérations de transtypage rend inutile la vectorisation d'expression mélangeant plusieurs types de données.
- L'expression doit utiliser des opérateurs ou des fonctions vectorisables. Certaines fonctions fournies par E.V.E. n'ont en effet aucun équivalent vectoriel⁴.

Ces contraintes sont clairement issues des limitations de l'implantation. Leur non-respect rend impossible la vectorisation d'une expression. En pratique, deux

⁴comme par exemple les fonctions de manipulation de forme ou certaines fonctions issues de `math.h`

cas se présentent : soit l'expression vérifie l'ensemble de ces contraintes et E.V.E. génère un code utilisant les primitives ALTIVEC, soit l'expression viole au moins une de ces contraintes et E.V.E. génère un code utilisant des fonctions scalaires classiques. Pour des raisons de performances, il est bien évident que la vérification de ces contraintes de vectorisation doit être effectuée bien avant l'exécution effective du calcul. Ceci est possible grâce à la métaprogrammation *template*. Nous allons ainsi pouvoir définir un ensemble de règles qui vont être résolues à la compilation. Le résultat de cette évaluation est ensuite utilisé dans une construction de type `if ... else` statique qui oriente le compilateur vers une des deux variantes du générateur de code. Les éléments nécessaires à l'évaluation de cette condition statique sont intégrés au différentes classes et fonctions *templates* de création d'expressions. Leur définition et leur implantation se fait de manière naturelle et suivant le processus de création des expressions présenté dans la section 4.3.2.

En premier lieu, il convient de déterminer si une instance de la classe `array` est vectorisable. Une instance de la classe `array` est vectorisable si sa liste d'options contient le marqueur `simd` et si le type de ses éléments est nativement supporté par ALTIVEC . Ce calcul est décrit dans le listing 4.25.

Listing 4.25 – Calcul de «vectorisabilité» d'une instance de `array`

```

1 template<class T, class O> class array
2 {
3     // ...
4     typedef typename opt_filter<O>::type_t      opt_t;
5     typedef typename type_at<opt_t,O>::type_t    simd_t;
6     typedef typename simd_t::type_t              vec_t;
7     typedef boxed<vec_info<T>::Length != 1>    sup_t;
8     typedef logical_and<vec_t,sup_t>            cond_t;
9     typedef typename cond_t::type_t             vect_t;
10
11    // ...
12 }
```

L'évaluation commence ligne 4 par le recouvrement de la liste d'options triée et épurée de ses éventuels doublons. Nous recouvrons l'état de l'option `simd` aux lignes 5 et 6. A ce moment, le type `vec_t` contient un indicateur de type booléen (cf annexe I). Nous continuons ensuite le calcul de vectorisation en récupérant à la ligne 7 un indicateur de type booléen déterminant si le type `T` est supporté par

ALTIVEC. Pour cela, nous utilisons une classe `vec_info` qui renvoie via sa valeur `Length` la largeur d'un vecteur de ce type. Dans le cas général, cette valeur vaut 1. Dans le cas d'un type supporté par ALTIVEC, cette valeur vaut 4, 8 ou 16 selon le type. Le test effectué génère alors un type `true_t` ou `false_t` qui correspond bien au fait que `T` soit supporté par ALTIVEC. Enfin, ligne 8, nous effectuons la combinaison des deux indicateurs de type précédemment construit en utilisant une version statique du ET logique (cf annexe I). Au final le type `vect_t` s'évalue comme le type `true_t` ou `false_t` reflétant la vectorisabilité de l'instance de `array`.

Un mécanisme identique est utilisé au sein de chaque classe composant les éléments d'un arbre de syntaxe abstraite au sein de E.V.E. . Un mécanisme récursif permet ensuite à chacun de ses éléments d'interroger l'état de «vectorisabilité» de ses éventuels fils afin d'évaluer son propre état. Le listing 4.26 présente par exemple l'évaluation de cet état pour un nœud contenant un opérateur binaire. Rappelons encore que l'ensemble de ce calcul est effectué par le compilateur au moment où il instancie les diverses définitions de types et les différents *templates* qui composent ce test.

Listing 4.26 – Evaluation de la «vectorisabilité» d'un nœud binaire

```

1 template<class L, class R, template<class, class> class F>
2 class binary_node
3 {
4     // ...
5     typedef typename L::vect_t           l_t;
6     typedef typename R::vect_t           r_t;
7     typedef typename F::vect_t           f_t;
8     typedef logical_and<l_t, r_t>::type_t    v1_t;
9     typedef logical_and<v1_t, f_t>::type_t  vect_t;
10    // ...
11 };

```

Cette évaluation consiste à comparer les états des fils droit et gauche et à combiner ces résultats avec l'état de «vectorisabilité» du foncteur associé au nœud. Comme chaque élément de l'arbre de syntaxe abstraite expose son propre type `vec_t`, l'évaluation se fait de manière récursive jusqu'à rencontrer une feuille de type `array`. Si une des feuilles s'avère contenir une constante, elle est considérée vectorisable par défaut. Un mécanisme interne à la construction de telles feuilles permet en effet de transtyper en amont la valeur de la constante pour qu'elle cor-

responde au type de la feuille de type `array` la plus proche au sein de l’arbre de syntaxe.

La dernière étape consiste à sélectionner, en fonction du type `vec_t` de la racine d’une expression, le mode d’évaluation à utiliser au sein de l’opérateur d’affectation de `array` (listing 4.27). Cette sélection utilise de nouveau le métaméopérateur ET logique pour confronter l’état de l’expression et l’état de l’instance destination. Cette évaluation permet ensuite d’instancier le type `evaluator` de manière à utiliser sa spécialisation scalaire ou vectorielle.

Listing 4.27 – Vérification de la «vectorisabilité» à l’affectation

```

1 template<class XPR> template<class T, class O>
2 array<T,O>& array<T,O>::operator=( const XPR& src )
3 {
4     typedef typename XPR::vect_t           x_t;
5     typedef logical_and<x_t, vect_t>      cond_t;
6     typedef typename cond_t::type_t       res_t;
7     typedef evaluator<array<T,O>, XPR, res_t > eval_t;
8
9     eval_t::store(*this, src);
10    return *this;
11 }

```

Au final, l’appel à `evaluator::store` effectue l’évaluation effective de l’expression. Au sein de la méthode `store`, les appels successifs à la méthode `load_val` sont résolus de manière classique. La seule différence réside dans le code des différents foncteurs qui utilisent l’information portée par le type `vect_t` pour déterminer quelle fonction bas niveau appeler. Les foncteurs représentant les opérateurs des expressions sont pourvus d’une méthode supplémentaire — `Load_Chunk` — qui prend en charge l’évaluation vectorielle du calcul (listing 4.28). L’examen du code assembleur final montre lui aussi une forte similarité avec le code assembleur obtenu par la compilation d’un code C ALTIVec équivalent.

Cette méthode permet donc de déterminer à la compilation la meilleure stratégie de parallélisation possible pour une expression. Le code complet intègre en outre des tests équivalents pour déterminer de manière statique divers paramètres d’optimisations comme le facteur de déroulage ou les stratégies de pré-chargement.

Listing 4.28 – Foncteur Func_Add : version vectorielle

```

1 template<class L, class R> struct Func_Add
2 {
3     typedef typename L::return_t l_t;
4     typedef typename R::return_t r_t;
5     typedef vec_type<l_t>::type_t el_t;
6
7     static inline return_t Load_Val( l_t l, r_t r )
8     { return l+r; }
9
10    static inline el_t Load_Chunk( el_t l, el_t r )
11    { return vec_add(l,r); }
12 };

```

Dans les cas où l'évaluation de l'expression nécessite une vectorisation spécifique ou un schéma de chargement spécifique, ce système d'évaluation est mis en défaut. Le seul moyen de limiter la dégradation des performances est alors de laisser le compilateur générer une instance temporaire de array ou de view et de lui permettre de décomposer l'évaluation totale de l'expression. Ainsi, le calcul présenté dans le listing 4.29 utilise la fonction de calcul de transposée de matrice qui ne peut être exprimée comme une opération vectorisable de la manière présentée ci-dessus :

Listing 4.29 – Exemple de calcul irrégulier

```

1 array<double> r,a,b,c;
2 r = (trans(a)+b)/c;

```

Son évaluation par le compilateur est effectuée de la manière suivante :

Listing 4.30 – Exemple de calcul irrégulier ré-évalué

```

1 array<double> r,a,b,c,tmp;
2 tmp = trans(a);
3 r = (tmp+b)/c;

```

Dans cette version, la transposée de a est pré-calculée et son évaluation temporaire est réinjectée dans une expression qui devient de fait évaluable vectoriellement. Si cette méthode semble en contradiction avec le principe même de E.V.E.,

il n'y a guère de choix pour assurer une écriture transparente de telles fonctions en conservant une interface homogène et assurer des performances globalement élevées. En effet, incorporer ces schémas d'accès irrégulier au sein du code d'évaluation générique permettrait certes d'éliminer la création de ces conteneurs temporaires mais aurait un impact non négligeable sur les performances de toutes les fonctions vectorisables de manière régulière. Cette solution met en évidence un problème récurrent dans la définition de tels outils. Pour permettre d'optimiser de façon efficace 95% des fonctions de E.V.E., il est nécessaire de sacrifier une partie des performances pour les 5% restant.

4.3.4 Optimisations spécifiques

Un des avantages de l'extension ALTIVEC est de fournir un certain nombre de primitives composites qui effectuent en une passe plusieurs opérations différentes. Parmi celles ci, on trouve par exemple la primitive `vec_madd`. L'appel à `vec_madd(a, b, c)` effectue le calcul de $a * b + c$. Son implantation ALTIVEC offre un gain significatif – de l'ordre de 80 % – par rapport à l'écriture explicite d'un produit suivi d'une somme. Au sein de E.V.E., il est donc intéressant de repérer les constructions du type $a * b + c$ et de les remplacer par l'application d'un opérateur ternaire virtuel `madd(a, b, c)`. L'idée de départ est de revenir sur l'arbre de syntaxe abstrait de l'expression $a * b + c$ (fig. 4.1). Grâce à cette représentation, il est possible de définir deux cas d'utilisation de l'opérateur + :

- Si l'opérateur + est utilisé sur deux opérandes quelconques, l'appel est résolu de manière classique.
- Si l'opérateur plus possède un opérande de type : `expression<binary_node <X, Y, Func_Mul> >` il convient de générer un nœud ternaire utilisant le foncteur `Func_MAdd`.

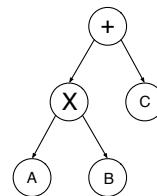


FIG. 4.1 – Arbre de syntaxe abstraite de $a * b + c$

Nous en venons donc à écrire une version spécialisée de l'opérateur + qui vient surcharger celle présentée dans le listing 4.23. Dans cette version, les éléments internes du nœud l sont extraits via les méthodes `left()` et `right()` et sont injectés au sein du nœud ternaire. Bien évidemment, une surcharge équivalente détecte les constructions de type $a + b * c$ et génère de la même manière une expression utilisant `vec_madd`. En utilisant la même stratégie, E.V.E. est capable de détecter l'ensemble des fonctions ALTIVEC composites et de générer le code adéquat.

Listing 4.31 – Surcharge spécialisée de `operator+` pour la gestion de `vec_madd`

```

1 template<class A, class B, class C>
2 expression<ternary_node<A,B,C,Func_MAdd> >
3 operator+(const expression<binary_node<A,B,Func_Mul>>& l
4 , const C& r )
5 {
6     typedef ternary_node<A,B,C,Func_MAdd> x_t;
7     return expression<expr_t>(x_t(l.left(),l.right(),r));
8 }
```

4.4 Évaluations des performances

Afin de valider notre approche, nous avons effectué des tests de performances permettant de mettre en avant l'apport de E.V.E. par rapport à une approche orientée objet naïve. Deux types de tests ont été menés : un test d'accélération relative et un test sur les performances des calculs mettant en œuvre de nombreux opérandes.

Il faut néanmoins tenir compte de la grande variabilité de telles mesures de performance [155] pour effectuer des tests significatifs. Pour s'affranchir de ce problème, notre protocole de test consiste à mesurer l'accélération d'un code utilisant des tableaux dont le nombre d'éléments assure leur chargement complet au sein du cache L1 du processeur, à savoir 1Ko à 32Ko, afin de limiter les pertes dues aux temps d'accès du cache L2 ou de la mémoire centrale. Enfin, afin de limiter les effets dus à la variation de la charge système l'ensemble de ces mesures ont été moyennée sur 10000 exécutions.

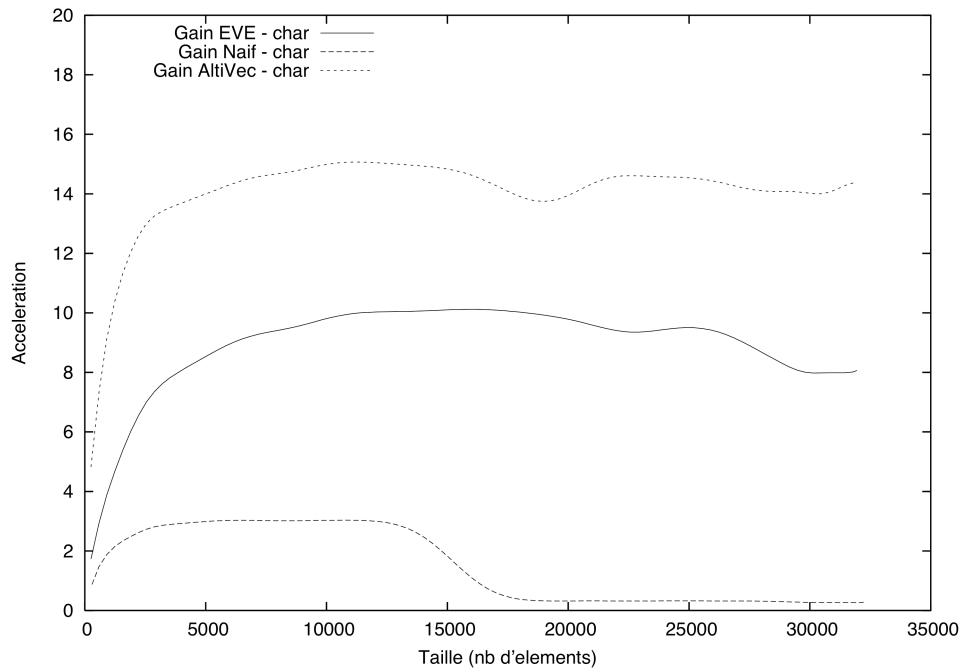


FIG. 4.2 – E.V.E. — Opérations simples sur des entiers 8 bit

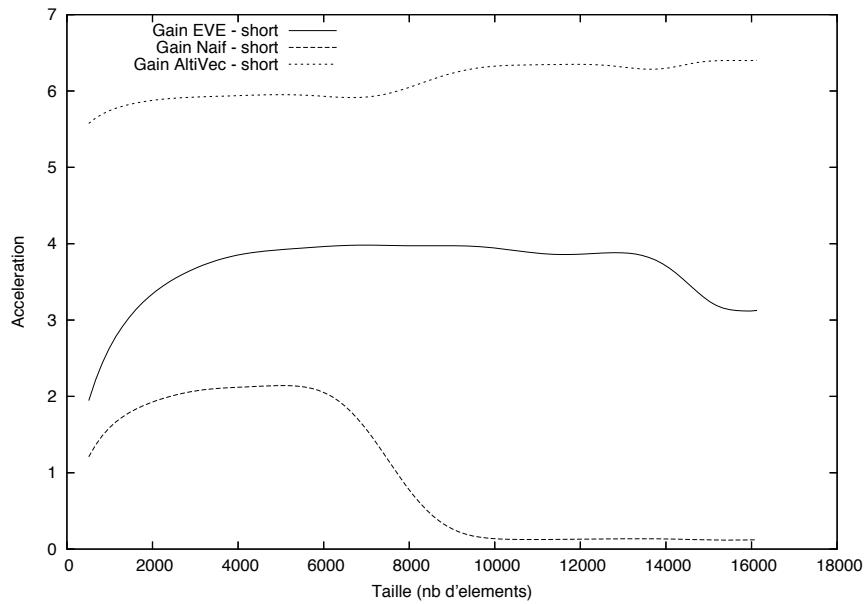


FIG. 4.3 – E.V.E. — Opérations simples sur des entiers 16 bit

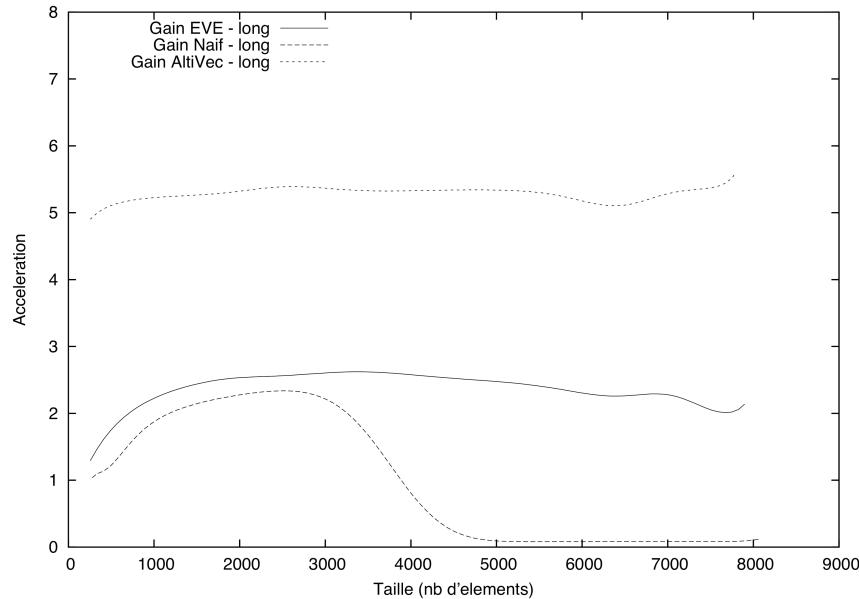


FIG. 4.4 – E.V.E. — Opérations simples sur des entiers 32 bit

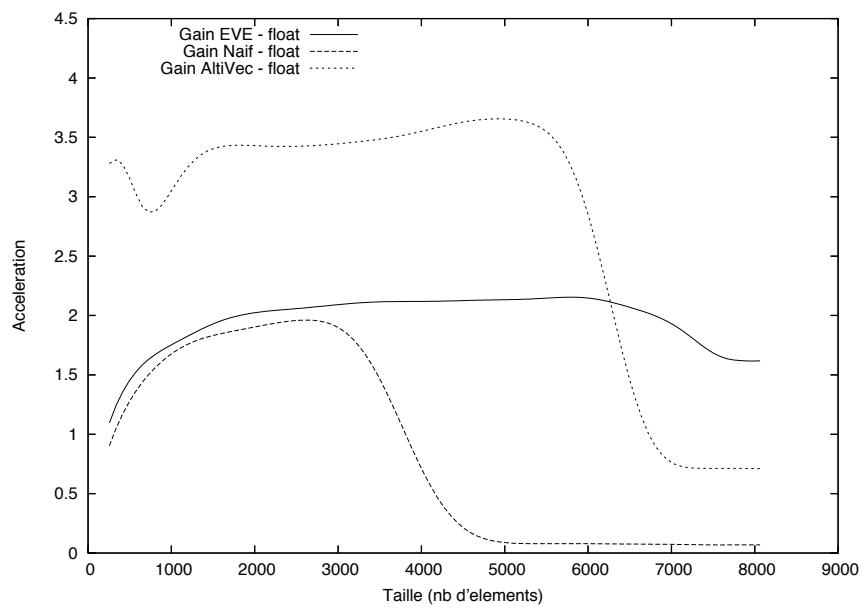


FIG. 4.5 – E.V.E. — Opérations simples sur des réels simple précision

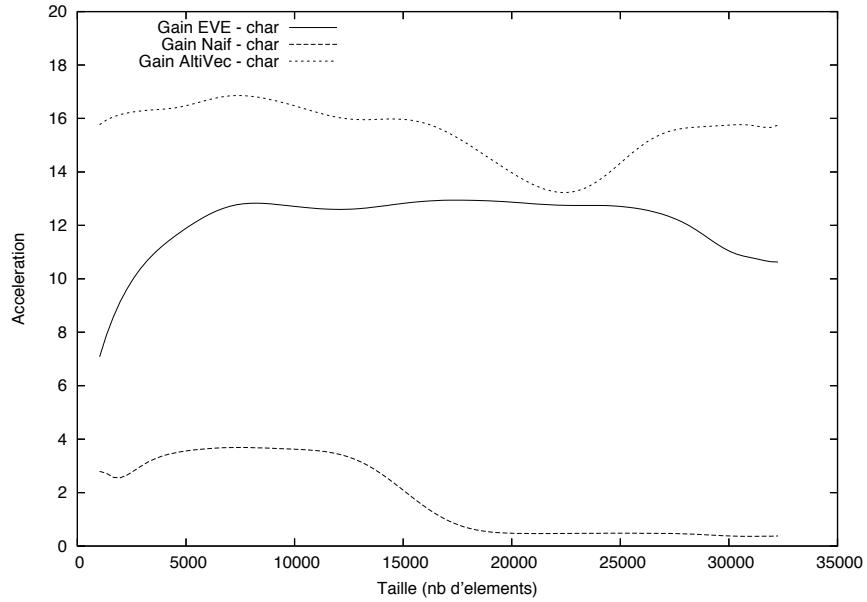


FIG. 4.6 – E.V.E. — Opérations composites sur des entiers 8 bit

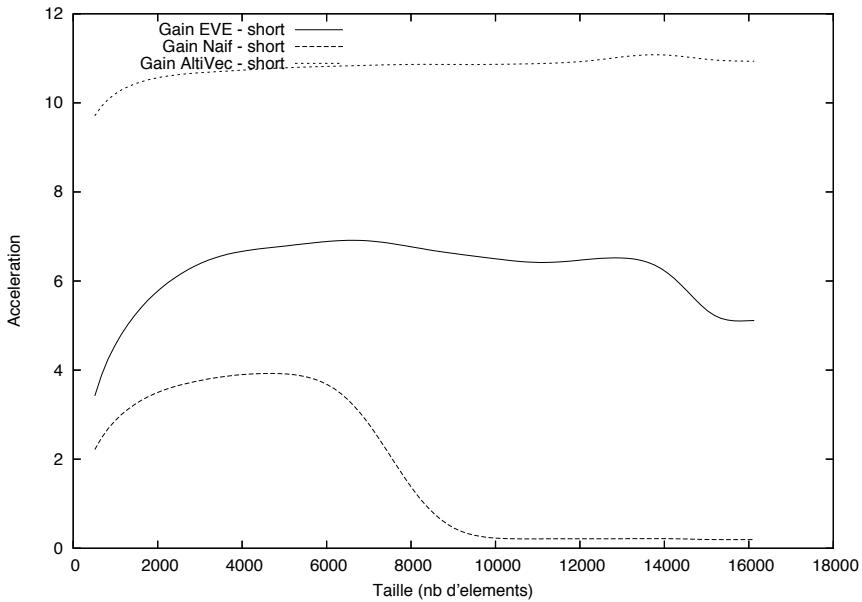


FIG. 4.7 – E.V.E. — Opérations composites sur des entiers 16 bit

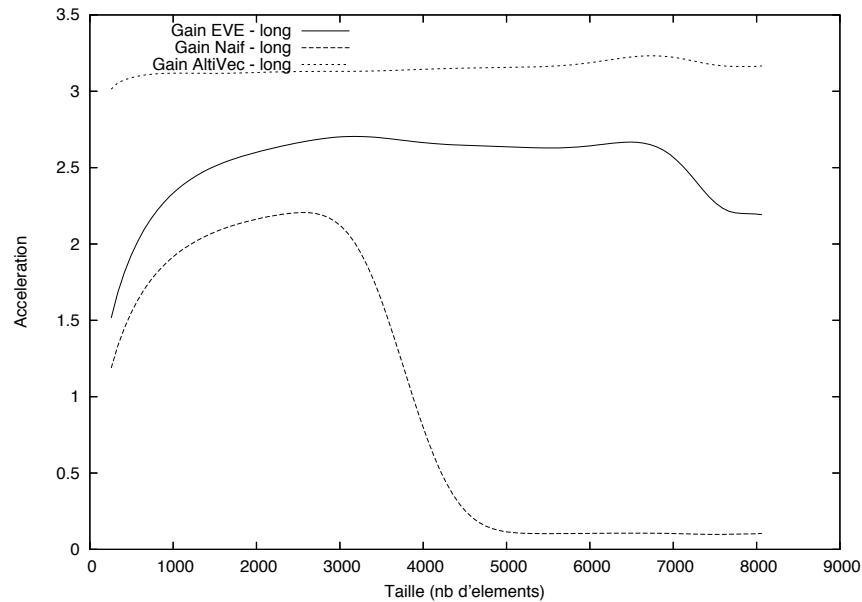


FIG. 4.8 – E.V.E. — Opérations composites sur des entiers 32 bit

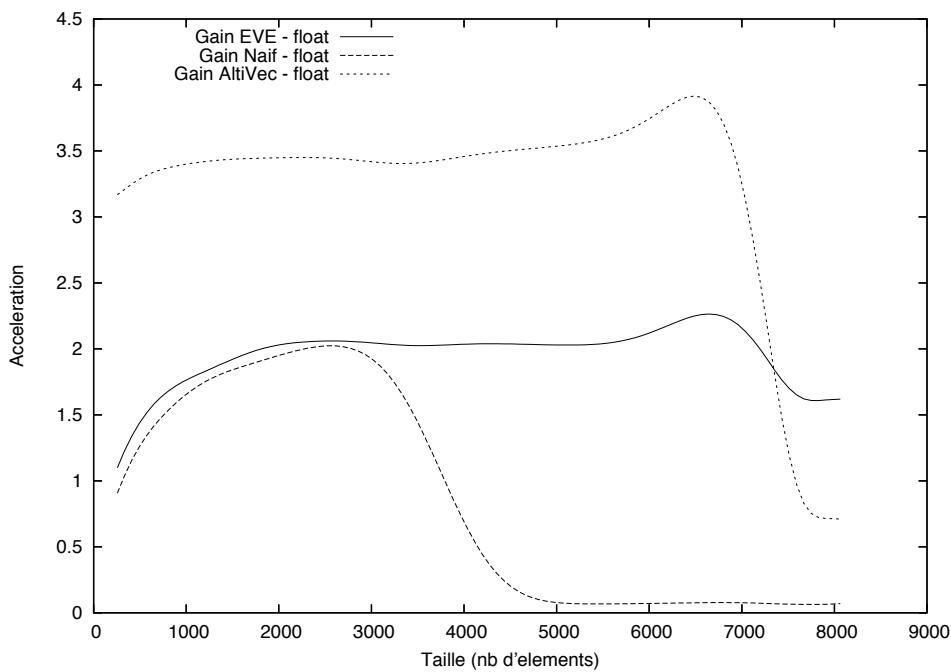


FIG. 4.9 – E.V.E. — Opérations composites sur des réels simple précision

4.4.1 Accélération SIMD

Les tests d'accélération consistent à mesurer trois accélérations relatives : celle d'un code C utilisant ALTIVEC , celle d'un code C++ accédant à ALTIVEC via une interface naïve et celle d'un code C++ utilisant E.V.E. par rapport à un code C séquentiel. Ces mesures portent sur deux types de fonctions :

1. **Des fonctions simples** : ces tests concernent l'ensemble des fonctions et opérateurs fournies par E.V.E. qui correspondent à l'appel d'une seule primitive ALTIVEC – en l'occurrence `vec_add` – et permettent d'estimer la surcharge due au générateur de code à base d'*Expression Templates*.
2. **Des fonctions composites** : ces tests concernent les fonctions et opérateurs fournis par E.V.E. qui utilisent des fonctions synthétiques – en l'occurrence une multiplication – permettant de pallier les manques du jeu d'instruction d'ALTIVEC, ceci afin de démontrer que ces ajouts se font sans perte de performances.

Les résultats de ces tests sont présentés sur les graphes 4.2 et 4.9 pour les quatre types de données disponibles en ALTIVEC — à savoir `char`,`short`,`long` et `float`, fournissant respectivement un gain maximal théorique de 16,8 et 4. Ils mettent en avant trois phénomènes principaux :

- Les accélérations fournies par l'approche orientée objet naïve ne dépassent pas 20 % de celles obtenues avec une implantation utilisant la seule interface C de l'extension ALTIVEC. On note aussi une chute drastique de ces gains avec l'augmentation de la taille du tableau de données. Ce phénomène montre bien l'impact des allocations et libérations de la zone mémoire des tableaux temporaires. Au delà de 8 à 16 Ko, le gain de l'implantation naïve ne dépasse pas 2 %, la rendant complètement inefficace.
- Les gains fournis par E.V.E. représentent entre 50 et 75 % du gain fournie par l'implantation C directe dans le cas des appels simples. Ils représentent 75 à 90 % de ce même gain dans le cas des tests sur les appels composites. Ces résultats confirment la pertinence de l'approche à base de *template*. Ils mettent aussi en avant le fait que les performances d'E.V.E. et d'ALTIVEC sont optimales lorsque le pipeline de l'unité ALTIVEC est correctement rempli. Nous verrons par la suite (cf section 4.5) que ce phéno-

mène participe grandement aux performances de E.V.E. et qu'il convient d'en tenir compte lors de l'écriture d'un code vectoriel.

- Le gain fourni par E.V.E. dépasse occasionnellement le gain de l'implantation C. Ces phénomènes proviennent de la manière dont le générateur de code de E.V.E. implante le parcours des tableaux source et destination. Ces parcours de tableaux sont effectués de manière à recouvrir le temps de chargement des données vers les registres vectoriels par le temps de gestion de la boucle et les temps de calcul.

4.4.2 Optimisation des compositions d'opérateurs

Ces tests ont pour but de montrer que la composition des fonctions et des opérateurs au sein de E.V.E. conserve des performances élevées, contrairement à l'approche objet naïve. Pour ce faire, nous avons procédé à des mesures de performances sur la composition d'une fonction simple – une addition – sur un nombre croissant d'opérandes de taille fixes, à savoir des tableaux de réels simple précision de 1024×1024 éléments (listing 4.32).

Listing 4.32 – Fragment du code source des tests de composition

```

1 array<float, settings< simd >> a(ofSize(1024,1024));
2 array<float, settings< simd >> r(ofSize(1024,1024));
3 array<double> time(ofSize(1,32));
4
5 // 1 Operateur +
6 tic();
7 for(int i k=0;k<1000;k++) r = a + a;
8 time(0) = toc(false);
9
10 // ...
11
12 // 32 Operateurs +
13 tic();
14 for(int i k=0;k<1000;k++)
15     r = a + a + a + a + a + a + a + a + a + a +
16             a + a + a + a + a + a + a + a + a +
17             a + a + a + a + a + a + a + a + a +
18             a + a + a;
19 time(5) = toc(false);

```

Les résultats de ces tests sont donnés dans le tableau 4.2 et corroborent l’hypothèse que les allocations mémoires dues aux variables temporaires de l’approche naïve sont la vraie source de pertes. En outre, on notera que les performances de E.V.E. diminuent de manière asymptotique pour se stabiliser vers 50% de la vitesse du code C équivalent. Cette baisse de performance provient principalement des diverses opérations annexes nécessaires au déroulement des calculs comme par exemple les vérifications de taille des tableaux et du fait que le code *template* généré augmente sensiblement la taille de l’exécutable final.

N_{op}	1	2	4	8	16	32
Ratio C++ naïf/C	10.3%	7.3%	5.8%	5.1%	3.7%	2.3%
Ratio E.V.E. /C	84.5%	78.4%	56.1%	48.3%	50.3%	50.2%

TAB. 4.2 – Effet du nombre d’opérandes sur les performances de E.V.E..

Ces tests préliminaires nous permettent de valider les différentes techniques utilisées par E.V.E. pour générer du code profitant de l’accélération fournie par ALTIVEC. Afin de conclure ces tests, nous proposons d’implanter une application réaliste qui nous permettra de montrer que l’expressivité de E.V.E. est suffisante pour exprimer des problèmes non triviaux et comment les choix de modèles d’optimisation permettent d’obtenir des performances satisfaisantes.

4.5 Étude de cas

L’étude de cas que nous allons mener dans ce chapitre a pour but de démontrer comment l’implantation d’un algorithme et sa vectorisation sont réalisées à l’aide de E.V.E. et d’évaluer les accélérations obtenues entre l’implantation séquentielle et l’implantation SIMD de cet algorithme. Les performances de E.V.E. par rapport à des implantations ALTIVEC développées de manières *ad hoc* ayant été démontrées au paragraphe précédent, nous nous attarderons sur l’aspect expressivité de E.V.E. et nous montrerons que les performances obtenues sont très satisfaisantes pour une bibliothèque utilisant un tel niveau d’abstraction.

4.5.1 Présentation de l’algorithme

Une des problématiques classiques du traitement d’image consiste à déterminer une manière correcte de représenter l’information de couleur — qui est une

propriété continue — dans un système informatique qui lui ne manipule qu'un espace de données discret. Il existe pour cela beaucoup de techniques. La plus utilisée est la représentation RGB, qui décrit une couleur par ses composantes rouges, vertes et bleues. Cette représentation correspond exactement à l'affichage des couleurs sur un écran par projection d'un flux d'électrons plus ou moins intense sur des pastilles rouges, vertes et bleues, qui en se mélangeant donnent l'impression de couleur.

La représentation YUV est aussi très utilisée, principalement dans tout ce qui est compression d'image. Y représente la luminance de la couleur, et U et V, la chrominance de cette couleur dans le rouge et le bleu. Cette représentation est utile, car l'oeil est plus sensible aux variations de luminances qu'aux variations de chrominance. Séparer ces trois composantes permettra donc de pouvoir dégrader plus les chrominances, tout en conservant mieux la luminance. De nombreux algorithmes utilisent cette représentation YUV pour opérer des détections dans l'espace des chrominances ou dans l'espace de la luminance. Néanmoins, les divers systèmes d'acquisition d'images ne permettent pas nécessairement d'obtenir une image directement au format YUV et il est alors nécessaire de convertir ces données. En première approche, nous pouvons estimer la luminance par la moyenne des trois composantes Rouge, Vert et Bleu. Cependant, cette approche ne tient pas compte de la sensibilité de l'oeil aux couleurs qui perçoit le vert comme beaucoup plus lumineux que le bleu.

A partir de cette constatation, on établit empiriquement les formules de conversions suivantes. Soit un $P_{rgb} = (R, G, B)$, un pixel encodé sous le format RGB et $P_{yuv} = (Y, U, V)$ sa conversion au format YUV. On a alors :

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

On utilise couramment une représentation en virgule fixe de cette formule afin de ne pas avoir à effectuer de transtypages.

Listing 4.33 – Pseudo-code pour la conversion RGB/YUV

```

1  Y = min(abs(-2104*R+4130*G+802*B+135168) >> 13, 235)
2  U = min(abs(-1214*R-2384*G+3598*B+1052672) >> 13, 240)
3  V = min(abs( 3598*R-3013*G-585*B+1052672) >> 13, 240)

```

Dans cette formulation, R, G et B sont typiquement codés sur 32 bits.

4.5.2 Implantations séquentielles

Nous présentons ici l'implantation séquentielle d'un algorithme de conversion d'une image RGB en trois images contenant les composantes Y, U et V de cette dernière. Dans cette implantation, l'image RGB de $h \times w$ pixels est représentée par un triplet de tableaux de $h \times w$ éléments de type entier 32 bits. Ce tableau contient alors de manière contiguë l'ensemble des valeurs Rouges de chaque pixel, puis les valeurs Vertes et les valeurs Bleus. les composantes Y, U et V de l'image converties sont stockées dans un tableau d'entiers 32 bits de $h \times w$ éléments. L'implantation séquentielle en C de l'algorithme est triviale : elle consiste à parcourir chaque partie de l'image RGB initiale et de calculer les valeurs YUV correspondantes via la formule en virgule fixe exposée plus haut.

Listing 4.34 – Conversion RGB/YUV — Version C séquentielle

```

1 void RGB2YUV_C(int* rgb, int* y, int* u, int* v, size_t sz)
2 {
3     int* R = rgb;
4     int* G = rgb+sz;
5     int* B = rgb+2*sz;
6
7     for(size_t i=0; i<sz; i++)
8     {
9         y[i] = min(abs( 2104*R[i] + 4130*G[i] +
10                     802*B[i] + 135168) >> 13, 235);
11         u[i] = min(abs(-1214*R[i] - 2384*G[i] +
12                     3598*B[i] + 1052672) >> 13, 240);
13         v[i] = min(abs( 3598*R[i] - 3013*G[i] -
14                     585*B[i] + 1052672) >> 13, 240);
15     }
16 }
```

Les temps d'exécution τ en milliseconde de cette implantation sont donnée dans le tableau 4.3 pour des images de $N \times N$ pixels avec $N \in \{128, \dots, 4096\}$.

N	128	256	512	1024	2048	4096
τ	1.27	6.32	33.15	134.00	531.42	2204.95

TAB. 4.3 – Temps d'exécution (ms) de RGB2YUV_C

La version séquentielle utilisant E.V.E. s'écrit elle aussi très simplement en

appliquant les formules données plus haut directement au tableau représentant l’images RGB (listing 4.35).

Listing 4.35 – Conversion RGB/YUV — Version E.V.E. séquentielle

```

1 void RGB2YUV_EVE (const array<int>& rgb, array<int>& y,
2                     array<int>& u, array<int>& v)
3 {
4     view<int> R = rgb.ima_view(0);
5     view<int> G = rgb.ima_view(1);
6     view<int> B = rgb.ima_view(2);
7
8     y = min(shr(abs(2104*R+4130*G+
9                     802*B+135168),13),235);
10    u = min(shr(abs(-1214*R-2384*G+
11                  3598*B+1052672),13),240);
12    v = min(shr(abs(3598*R-3013*G-
13                  585*B+1052672),13),240);
14 }
```

Les principales différences entre cette version et la version C sont :

- L’utilisation de `view` permet d’accéder simplement à des sous-tableaux contigus d’une autre instance de `array` via la méthode `ima_view`.
- La fonction `shr` qui remplace l’opérateur de décalage de bit.
- La boucle `for` et les accès par index aux valeurs des différents tableaux ont entièrement disparu, rendant l’écriture du calcul de `y` très proche de sa formulation mathématique.

Les temps d’exécution τ en millisecondes et le rapport ϵ au temps d’exécution de la version C de cette implantation sont donnés dans le tableau 4.4. On note que le surcoût du au mode séquentiel de E.V.E. reste inférieure à 30 % et devient quasiment négligeable lorsque la taille des images augmente.

Nous allons maintenant voir comment les options d’optimisations de E.V.E. permettent d’accélérer l’exécution de cet algorithme.

N	128	256	512	1024	2048	4096
τ	1.80	8.66	34.61	137.05	545.53	2244.79
ϵ	70.5 %	72.9 %	95.8 %	97.7 %	97.4 %	98.2 %

TAB. 4.4 – Temps d'exécution de RGB2YUV_EVE

4.5.3 Implantations optimisées

Deux types d'optimisations sont envisageables. Tout d'abord, nous allons activer le mode SIMD de E.V.E. et permettre à notre implantation de bénéficier de l'accélération de l'extension ALTIVec. Nous allons ensuite mettre en place une stratégie de déroulage partiel. Pour ces deux types d'options, nous mesurerons le gain par rapport à la version C initiale.

4.5.3.1 Implantation SIMD

Le code source de la version SIMD de notre algorithme est présenté dans le listing 4.36. Comme énoncé dans le paragraphe 4.2.2, l'activation des optimisations SIMD est extrêmement simple et se résume à l'ajout du paramètres `simd` au sein de la liste d'option des conteneurs.

Listing 4.36 – Conversion RGB/YUV — Version E.V.E. SIMD

```

1 void RGB2YUV_SIMD (const array<int>& rgb, array<int>& y,
2                         array<int>& u ,array<int>& v)
3 {
4     view<int, settings<simd>> R=rgb.ima_view(0);
5     view<int, settings<simd>> G=rgb.ima_view(1);
6     view<int, settings<simd>> B=rgb.ima_view(2);
7
8     y = min(shr(abs(2104*R+4130*G+
9                     802*B+135168),13),235);
10    u = min(shr(abs(-1214*R-2384*G+
11                  3598*B+1052672),13),240);
12    v = min(shr(abs(3598*R-3013*G-
13                  585*B+1052672),13),240);
14 }
```

Les temps d'exécution τ en millisecondes et l'accélération Γ_C de cette implantation par rapport à la version séquentielle écrite en C sont données dans le tableau

4.5.

N	128	256	512	1024	2048	4096
τ	0.37	1.82	9.34	38.89	153.72	609.13
Γ_C	3.43	3.46	3.54	3.44	3.46	3.62

TAB. 4.5 – Temps d'exécution (ms) de RGB2YUV SIMD

L'accélération fournie par E.V.E. dans cet algorithme complexe est très satisfaisante — de l'ordre de 85% du gain maximum théorique (ici 4) pour ce type de donnée — et reste relativement stable pour toutes les tailles d'images.

4.5.3.2 Implantation SIMD avec déroulage

Pour utiliser un déroulage partiel de l'évaluation de ces calculs, il nous faut tenir compte de l'architecture de l'extension ALTIVec et nous intéresser aux unités internes de calculs mises en jeu. Ici, l'ensemble des fonctions utilisées fait partie de l'unité entière d'ALTIVec à l'exception de `abs`, ce qui nous conduit à utiliser un pas de déroulage de 5. Ce pas correspond en effet à la profondeur du pipeline de l'unité entière composite qui sera chargée d'exécuter la fonction `abs`. Le code source de la version SIMD déroulée de notre algorithme est présenté dans le listing 4.37. De la même manière que précédemment, nous modifions les options d'optimisation des conteneurs en utilisant le marqueur `unroll` pour spécifier le niveau de déroulage souhaité.

Les temps d'exécution τ en millisecondes et le gain Γ_C de cette implantation par rapport à la version séquentielle écrite en C sont donnés dans le tableau 4.6.

N	128	256	512	1024	2048	4096
τ	0.30	1.49	7.64	32.21	126.01	500.03
Γ_C	4.23	4.25	4.33	4.16	4.22	4.41

TAB. 4.6 – Temps d'exécution (ms) de RGB2YUV USIMD

Dans cette implantation, le bénéfice du déroulage est très notable et permet, comme nous l'avions pressenti au paragraphe 2.6.1, de dépasser le gain purement du à la nature SIMD de l'extension ALTIVec et de profiter de la profondeur de son pipeline.

Listing 4.37 – Conversion RGB/YUV — Version E.V.E. SIMD avec déroulage

```

1 void RGB2YUV_USIMD (const array<int>& rgb, array<int>& y,
2                         array<int>& u ,array<int>& v)
3 {
4     view<int, settings<simd, unroll<5>>> R=rgb.ima_view(0);
5     view<int, settings<simd, unroll<5>>> G=rgb.ima_view(1);
6     view<int, settings<simd, unroll<5>>> B=rgb.ima_view(2);
7
8     y = min(shr(abs(2104*R+4130*G+
9                     802*B+135168 ),13),235);
10    u = min(shr(abs(-1214*R-2384*G+
11                  3598*B+1052672),13),240);
12    v = min(shr(abs(3598*R-3013*G-
13                  585*B+1052672),13),240);
14 }
```

4.5.4 Discussion

Trois points ressortent de cette étude de cas. Tout d'abord, le passage d'un code C séquentiel à un code séquentiel utilisant E.V.E. puis un code SIMD est très simple. La richesse de l'interface de E.V.E. et son modèle à base de tableaux permet une réécriture rapide des problèmes classiques du traitement du signal ou du traitement d'images.

Ensuite, la vectorisation est très aisée — elle se réduit à une simple spécification de marqueur de type — et donne de très bons résultats, proches de ceux que l'on pourrait obtenir par une vectorisation manuelle. La stratégie d'optimisation par déroulage mise en place est efficace mais non optimale. Une analyse poussée de ces stratégies est nécessaire, tout comme la mise en place d'un système plus complexe de déroulage.

Enfin, la mise en oeuvre d'optimisations plus poussées nécessite encore une connaissance pointue de l'extension ALTIVec. Dans un cas général, le développeur peu au fait de ces problèmes va devoir tâtonner pour déterminer un pas de déroulage convenable. Il serait alors envisageable de trouver un moyen de déléguer à E.V.E. elle-même la tâche de déterminer ce pas de déroulage en analysant les expressions qu'elle évalue.

4.6 Conclusion

La définition d'outils performants pour la programmation d'une machine comme BABYLON n'est pas une tâche aisée. L'analyse des différents outils existants a montré qu'il était difficile de fournir une implantation à la fois performante et de haut niveau et spécialement dans le cas du développement de bibliothèques dédiées. Le développement de ce type d'outil se heurte en général aux limitations du langage cible et ne peut garantir les performances finales des applications. Nous avons néanmoins montré qu'un tel développement est possible en utilisant un aspect relativement peu exploité dans le cadre du calcul scientifique : la métaprogrammation *template*.

E.V.E. tire pleinement partie de ces techniques et permet donc d'utiliser l'extension ALTIVEC dans le cadre d'un modèle de programmation simple et familier aux les développeurs de la communauté Vision. Ses principaux apports au vu des modèles et outils de développement SIMD existants sont :

- Un modèle de programmation basé sur la manipulation de tableaux numériques à plusieurs dimensions grâce auquel la transition entre les modèles mathématiques et algorithmiques communément employés dans la communauté Vision et leur expression en tant que programme C++ est facilitée. E.V.E. propose en effet une interface très proche de celle de MATLAB® qui est un outil largement diffusé dans la communauté Vision pour permettre le prototypage rapide d'algorithmes et permet donc de réutiliser l'ensemble des solutions déjà développées grâce à cet outil de manière très rapide.
- Une implantation efficace qui permet de s'abstraire des limitations des implantations orientées objet classiques en C++. Cette implantation permet de tirer partie d'une large fraction de l'accélération de l'extension ALTIVEC et fournit un large panel d'options d'optimisations annexes.

E.V.E. démontre ainsi que l'utilisation d'ALTIVEC peut se faire dans le cadre d'un modèle de programmation simple et maîtrisé.

Chapitre 5

La bibliothèque QUAFF

«*If you can't get rid of the skeleton in your closet,
you'd best teach it to dance.*»
George Bernard Shaw

Contrairement aux outils de programmation dédiés aux extensions SIMD, la programmation structurée des machines MIMD via MPI a fait l'objet de nombreuses études. Parmi les diverses approches existantes, notre choix s'est porté sur les approches basées sur les squelettes algorithmiques. Plusieurs bibliothèques construites selon cette approche ont été proposées [45, 55, 22, 157, 119, 44, 54], chacune définissant son propre jeu de squelettes et un modèle de programmation idoine. Nous aurions très bien pu utiliser une de ces bibliothèques — MUESLI, Lithium ou eSkel par exemple — et directement tirer parti de l'approche squelette sans chercher plus avant. Néanmoins, aucune d'entre elles ne satisfait pleinement nos objectifs conjoints de performance et d'abstraction. Nous nous proposons donc de définir et d'implanter une telle bibliothèque — la bibliothèque QUAFF¹ [67] — qui, comme E.V.E., utilise des techniques d'implantation avancées afin de garantir à la fois un niveau d'abstraction élevé et de bonnes performances.

Nous commencerons par présenter le modèle de programmation sur lequel repose QUAFF. Nous nous attardons ensuite sur l'interface effectivement fournie par QUAFF aux développeurs et nous abordons les problématiques spécifiques de son implantation en évaluant les limitations d'une implantation classique et en proposant des techniques d'évaluation partielle permettant de réduire leurs effets.

¹*QUick Application from Flow-based Framework*

Nous terminons en évaluant les performances de QUAFF.

5.1 Modèle de programmation

Considérons une application parallèle définie par un **Graphe de Processus communiquant** (figure 5.1). Cette représentation permet de spécifier les schémas de communications entre les processus P_i , de mettre en évidence les fonctions séquentielles F_i utilisées au sein de cette application et d'expliciter les processus nécessaires au déroulement parallèle de l'application (ici les fonctions **Distrib** et **Collect**).

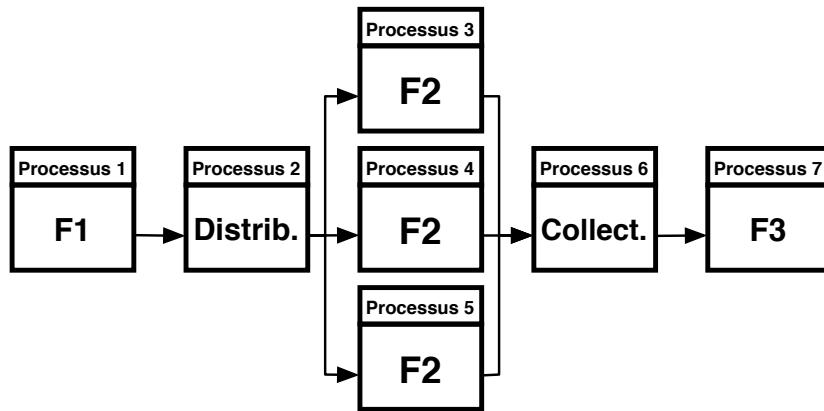


FIG. 5.1 – Graphe de processus communiquant d'une application parallèle

Dans ce graphe, on distingue deux parties distinctes (figure 5.2).

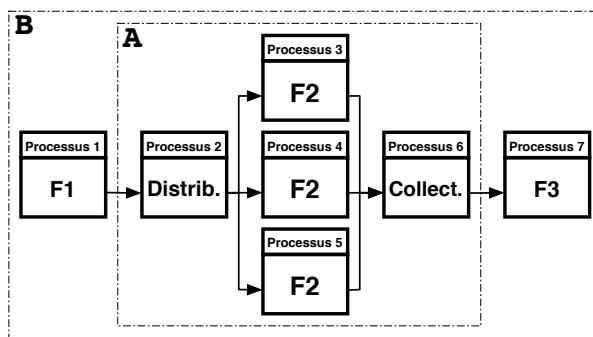


FIG. 5.2 – Hiérarchisation d'un Graphe de Processus communiquant

Dans le cas présenté, on trouve :

- un système d'équilibrage de charge (**A**) qui utilise une réserve de k processus auxquels sont transmises les données à traiter via une fonction (F_2). Lorsqu'une donnée est transmise au processus hébergeant la fonction **Distrib**, elle est transmise au premier processus esclave libre. Ce processus effectue la fonction F_2 et renvoie le résultat au processus hébergeant la fonction **Collect**. Pendant ce temps, la fonction **Distrib** continue de distribuer les données du flux d'entrée sur d'autres processus libres;
- un mécanisme de contrôle (**B**) qui consiste à exécuter une série de fonctions (ici F_1 , F_3 et la section (**A**)) sur le flux de données d'entrées de l'application. Chacune de ces fonctions est placée sur un processus différent et les résultats intermédiaires transitent de processus à processus à chaque fois qu'une fonction se termine. Si l'on considère un ensemble de fonctions (F_0, \dots, F_n) et un ensemble de processus (P_0, \dots, P_n) , l'exécution de cette structure consiste à recevoir sur le processus P_i des données en provenance du processus P_{i-1} , exécuter la fonction F_i sur ces données, transmettre le résultat de ce calcul au processeur P_{i+1} et finalement se préparer à recevoir de nouvelles données du processeur P_{i-1} . En régime permanent, les fonctions (F_0, \dots, F_n) s'exécutent en parallèle. On reconnaît donc ici une structure de type *pipeline*.

Ce type de construction se révèle extrêmement courant dans la pratique de la programmation parallèle. En fait, et comme nous l'avons vu au paragraphe 3.2, les modèles de programmation à base de squelettes algorithmiques reposent sur l'extraction de tels schémas récurrents. On définit ainsi un squelette comme un schéma générique, paramétré par une liste de fonctions et qu'il est possible d'instancier et de composer. Fonctionnellement, les squelettes algorithmiques sont des **fonctions d'ordre supérieur**, c'est à dire des fonctions prenant une ou plusieurs fonctions comme arguments et retournant une fonction comme résultat.

Pour le développeur, l'utilisation de ces squelettes permet de s'abstraire des considérations complexes de placement et d'ordonnancement et de manipuler des concepts de haut niveau [45, 163]. Définir une application parallèle revient alors à :

- instancier des squelettes en spécifiant les fonctions qui les définissent;
 - exprimer la composition des ces squelettes.
-

L'expression de la compositions peut se faire en encodant cette dernière sous la forme d'un **arbre** dont les nœuds représentent les squelettes utilisés et les feuilles, les fonctions séquentielles passées en paramètres à ces squelettes. Ainsi, la définition de l'application présentée sur la figure 5.1 se résume au schéma 5.3.

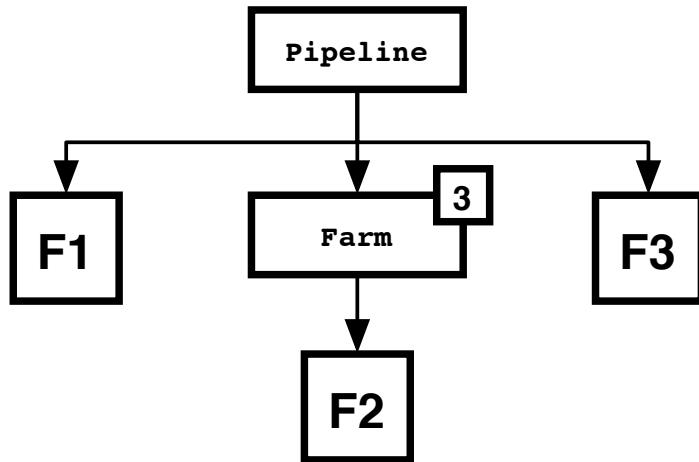


FIG. 5.3 – Réécriture sous forme de squelette d'une application parallèle

Dans ce schéma, le squelette **Pipeline** décrit le schéma générique correspondant à la section **B** du graphe de processus communiquant initial. Le squelette **Farm** représente quant à lui la partie **A** de ce même schéma. Les fonctions F_i apparaissent aux feuilles de l'arbre, c'est à dire en argument des squelettes. On note aussi que les fonctions **Distrib** et **Collect** n'apparaissent plus explicitement, car elles font partie intégrante du squelette **Farm**.

Cette représentation met en avant un des aspects les plus importants de l'approche à base de squelettes algorithmiques : à partir d'un nombre restreint de squelettes (classiquement moins d'une dizaine), il est possible de définir des applications complexes. Ceci suppose toutefois que l'on ait formalisé le type d'application que l'on va chercher à paralléliser, de définir précisément le jeu de squelettes que l'on désire mettre à disposition du développeur et de spécifier leurs sémantiques fonctionnelles et opérationnelles². On trouve dans la littérature plusieurs jeux de squelettes algorithmiques répondant à divers besoins [45, 22, 5, 88, 119, 54]. Au sein de QUAFF, les squelettes proposés se répartissent en trois

²La description de la sémantique opérationnelle des squelettes restera ici peu formalisée bien que des travaux comme [7] proposent une notation à base de règles de transition pour exprimer de manière formelle une telle sémantique.

groupes : les squelettes dédiés au parallélisme de contrôle, les squelettes dédiés au parallélisme de données et les squelettes dédiés à la structuration séquentielle de l'application.

5.1.1 Définition formelle d'une application

Un modèle classique d'application est proposé par Aldinucci et Danelutto [5, 6]. Il consiste à définir une application comme une fonction ψ de type³ :

$$\psi : \alpha \rightarrow \beta$$

qui est appliqué successivement sur les éléments d'un flux de données S constitué d'éléments x_i de type α :

$$S = \langle x_0, \dots, x_m \rangle$$

afin de produire un flux de données S' constitué d'éléments y_i de type β^4 :

$$\begin{aligned} S' &= \langle y_0, \dots, y_m \rangle \\ &= \langle \psi x_0, \dots, \psi x_m \rangle \end{aligned}$$

Si l'on se donne une fonction **lift** qui effectue l'application d'une fonction f sur un flux $\langle x_0, \dots, x_m \rangle$:

$$\text{lift } f \langle x_0, \dots, x_m \rangle = \langle f(x_0), \dots, f(x_m) \rangle$$

on définit alors l'application \mathcal{A}_ψ associée à ψ par :

$$\mathcal{A}_\psi = \text{lift } \psi$$

en sorte que :

$$\mathcal{A}_\psi(S) = S'$$

En règle générale, il existe une liste de fonctions séquentielles $[f_0, \dots, f_n]$ telle que :

$$\psi = f_n \circ \dots \circ f_0 = \underset{i=0}{\overset{n}{\circ}} f_{n-i}$$

La parallélisation de l'application \mathcal{A}_ψ définie par cette liste de fonctions passe alors par l'imbrication de squelettes et leur application aux éléments de la liste $[f_0, \dots, f_n]$ qui définit \mathcal{A} .

³Nous utiliserons ici la notation ML qui consiste à définir une fonction par son nom, le type de son ou ses arguments et son type de retour. Ainsi une fonction f acceptant un argument de type α et renvoyant un résultat de type β se note : $f : \alpha \rightarrow \beta$

⁴Nous utiliserons ici la notation ML pour l'appel des fonctions. Ainsi, l'application d'une fonction f sur a se notera $f a$. Pour les fonctions à plusieurs arguments, la notation utilisée est $f a b$ l'appel $f(a,b)$.

5.1.2 Squelettes dédiés au parallélisme de contrôle

QUAFF propose deux squelettes pour le parallélisme de contrôle : **Pipeline**, qui correspond à la composition parallèle de fonction, et **Pardo** qui prend en charge la mise en place de schéma de parallélisation *ad hoc* [44].

5.1.2.1 Le squelette Pipeline

Pipeline modélise les situations dans lesquelles une liste de fonctions, qui doivent être exécutées successivement, est distribuée sur un ensemble de processus afin d'effectuer l'équivalent parallèle de la composition de fonction. Sa sémantique fonctionnelle est donnée par la relation:

$$\text{Pipeline } [f_0, \dots, f_n] = \text{lift } \mathcal{F}$$

avec :

$$\mathcal{F} = \bigcirc_{i=0}^n f_{n-i} \quad \text{et } \forall i \in [0 \dots n], f_i : \alpha_i \rightarrow \alpha_{i+1}$$

La sémantique opérationnelle d'un **Pipeline** appliqué sur une liste de n fonctions fait que l'exécution de la fonction f_i sur l'élément x_j du flux de données se déroule en parallèle de l'exécution des fonctions $f_{i'} \in [0, n]$, $i' \neq i$ sur les éléments $x_{j-(i'-i)}$ du flux de données. Le parallélisme exploité provient alors du fait que les évaluations des différentes fonctions composant le **Pipeline** sur des éléments différents du flux de données sont indépendantes. Son graphe de processus communicants est donnée sur la figure 5.4.

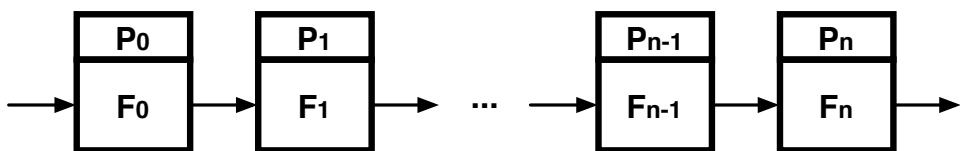


FIG. 5.4 – Graphe de processus communicants du squelette pipeline

Les grandeurs caractéristiques d'un **Pipeline** sont la latence et le débit. La latence est le temps nécessaire pour que le premier résultat de l'application du **Pipeline** soit disponible. Le débit mesure quant à lui le nombre de résultat fournis chaque seconde par le **Pipeline**. Considérons un **Pipeline** constitué de n fonctions $[f_0, \dots, f_n]$ et appliqué sur un flux $\langle x_0, \dots, x_m \rangle$. On évalue sa latence λ par la

relation :

$$\lambda = \sum_i \tau_i$$

où τ_i correspond au temps d'exécution de la fonction f_i . Le débit δ de ce **Pipeline** est quant à lui donné par la relation :

$$\delta = \min_{i \in [0, n]} \frac{1}{\tau_i}$$

5.1.2.2 Le squelette Pardo

Le squelette **Pardo** permet le placement *ad hoc* de N fonctions sur N processus sans spécifier de schéma de communication implicite mais en laissant le soin au développeur de préciser comment chacune de ces tâches va communiquer avec ses homologues. Cette notion de parallélisme *ad hoc* permet de faciliter l'expression d'applications complexes qui ne peuvent correspondre entièrement à un squelette ou à une combinaison des squelettes disponibles [44, 27]. Sa sémantique fonctionnelle s'exprime sous la forme :

$$\textbf{Pardo } [f_0, \dots, f_n] = \textbf{lift do } [f_0, \dots, f_n]$$

Avec :

$$\textbf{do } [f_0, \dots, f_n] x = [f_0 x, \dots, f_n x]$$

L'utilisation de **Pardo** est notamment nécessaire pour rassembler en une seule application un sous-ensemble d'applications opérant de manière indépendante sur le flux de données. Le parallélisme exploité provient alors du fait que chaque application s'exécute indépendamment des autres.

Le comportement temporel d'une instance de **Pardo** est alors défini par son temps d'exécution total. Si, pour **Pardo** $[f_0, \dots, f_n]$, on note τ_i le temps d'exécution de la fonction f_i , le temps d'exécution total de cette instance est donné par :

$$\tau_{pardo} = \max_{i \in [0, N]} \tau_i$$

5.1.3 Squelettes dédiés au parallélisme de données

Deux squelettes sont couramment utilisés pour la gestion du parallélisme de données : **Farm** qui gère les cas de parallélisme de données simples et **SCM**, qui gère des schémas de calcul par décomposition de domaines.

5.1.3.1 Le squelette Farm

Le squelette **Farm** modélise les situations où une fonction doit être répliquée afin de permettre de mettre en place un schéma d'équilibrage de charge. Pour cela, ce squelette utilise une réserve de k processus auquel il va transmettre les éléments du flux à traiter. Sa sémantique fonctionnelle est donnée par la relation :

$$\text{Farm } f = \text{lift } f$$

Du point de vue opérationnel, le parallélisme est exploité par le fait que l'évaluation de $f(x_i)$ est effectué en parallèle à celle $f(x_j)$ pour toute valeur de i différente de j . En terme d'implantation, lorsqu'une donnée est transmise au processus hébergeant le squelette **Farm**, elle est transmise au premier processus esclave libre. Ce processus exécute la fonction f et renvoie le résultat à un processus de collecte. Pendant ce temps, le processus de distribution continue à transmettre les données du flux d'entrée sur d'autres processus libres. Son schéma de placement/ordonnancement est donné sur la figure 5.5.

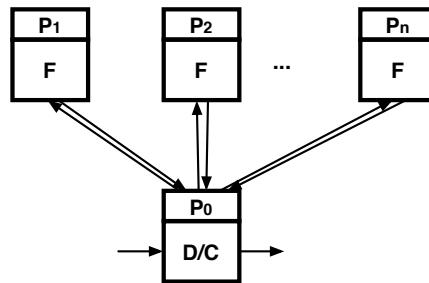


FIG. 5.5 – Schéma d'ordonnancement du squelette farm

La grandeur caractéristique du squelette **farm** est le temps de mise à disposition, c'est à dire le temps qui sépare l'entrée d'un élément du flux dans le squelette et la sortie du résultat correspondant. Sa valeur est donnée par la somme de τ_f , le temps d'exécution d'un processus esclave, de τ_{comm} , le temps de communication nécessaire pour envoyer la donnée d'entrée et recevoir le résultat. Dans le cas où tout les esclaves sont occupés, un temps supplémentaire τ_{wait} vient s'ajouter à ce temps de mise à disposition⁵ :

$$\tau_{disp} = \tau_f + \tau_{comm} + \tau_{wait}$$

⁵Dans le cas général, ces trois temps dépendent de la donnée x_i à traiter.

5.1.3.2 Le squelette SCM

Le squelette **SCM** permet d'utiliser un schéma fixe de décomposition des données correspondant à un parallélisme de données régulier. Sa sémantique fonctionnelle est donnée par la relation :

$$\mathbf{SCM} \ [\sigma, \phi, \mu] = \mathbf{lift\ scm} \ [\sigma, \phi, \mu]$$

Avec :

$$\begin{aligned}\mathbf{srm} \ [\sigma, \phi, \mu] \ x &= \mu (\mathbf{map} \ \phi \ (\sigma \ x)) \\ \mathbf{map} \ f \ [x_0, \dots, x_m] &= [f \ x_0, \dots, f \ x_m]\end{aligned}$$

Cette sémantique impose des contraintes sur les types des fonctions σ , ϕ et μ . Compte tenu de la définition, et le type de **srm**, on a⁶ :

$$\sigma : \alpha \rightarrow [\gamma_i] \quad \phi : \gamma_i \rightarrow \gamma_o \quad \mu : [\gamma_o] \rightarrow \beta$$

SCM effectue donc le traitement d'un élément x_i de type α en le décomposant en une liste d'éléments de type γ_i via la fonction σ . Cette liste est ensuite distribuée sur un ensemble de processus esclaves qui y appliquent la fonction ϕ . Les résultats de ces évaluations sont ensuite recouvrés sous forme d'une liste d'éléments de type γ_o qui est fusionnée par la fonction μ afin de reconstruire un élément de type β . Son schéma de placement/ordonnancement est donné sur la figure 5.6.

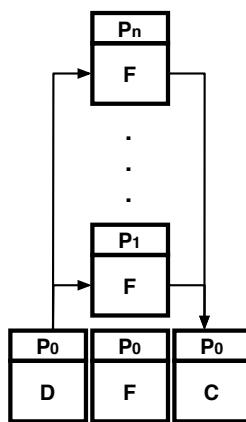


FIG. 5.6 – Schéma d'ordonnancement du squelette **srm**

La grandeur caractéristique d'un squelette **SCM** est le rapport entre le temps passé dans les communications nécessaires à la diffusion des sous-éléments et le temps de calcul effectif sur les processus esclaves.

⁶La notation $[\tau]$ représente le type «liste d'éléments de type τ ».

5.1.4 Squelettes dédiés à la structuration de l'application

Ces squelettes n'introduisent aucun parallélisme au sein d'une application mais permettent de structurer cette dernière en utilisant des constructions comme le branchement conditionnel ou la composition séquentielle de fonctions.

5.1.4.1 Le squelette Sequence

Sequence permet de composer séquentiellement un nombre arbitraire de tâche. Sa sémantique fonctionnelle est identique à celle du squelette **Pipeline** :

$$\text{Sequence } [f_0, \dots, f_n] = \text{lift } \mathcal{F}$$

avec :

$$\mathcal{F} = \bigcirc_{i=0}^n f_{n-i} \quad \text{et } \forall i \in [0 \dots n], f_i : \alpha_i \rightarrow \alpha_{i+1}$$

Pipeline et **Sequence** ne diffèrent que du point de vue opérationnel. En effet, au sein instance du squelette **Sequence**, seul le parallélisme inhérent à chaque fonction f_i est exploité lors de son exécution. Du point de vue temporel, la latence d'une **Sequence** est donné par la somme des temps d'exécution τ_i des fonctions qui f_i la composent.

$$\tau = \sum_i \tau_i$$

5.1.4.2 Le squelette Select

Le squelette **Select** est un équivalent de la construction `if ... then ... else` du langage C. Il est défini par un triplet de fonctions représentant respectivement la condition du test (τ), la fonction correspondante au cas *vrai* (ν) et la fonction correspondante au cas *faux* (ϕ). Sa sémantique fonctionnelle s'exprime comme :

$$\text{Select } [\tau, \nu, \phi] = \text{lift } \mathcal{F}$$

avec :

$$\mathcal{F}_x = \begin{cases} \nu x & si \quad \tau x = vrai \\ \phi x & si \quad \tau x = faux \end{cases}$$

Du point de vue opérationnel, **Select** se comporte en tout point comme une simple construction `if .. else` séquentielle.

5.1.5 Exemple de mise en œuvre

Considérons une application de traitement d'image consistant à appliquer un détecteur de contour sur les images provenant d'un flux vidéo. Sa définition met en œuvre un certain nombre de fonctions :

- **thresh** qui effectue le seuillage des images provenant du flux vidéo;
- **edge** qui extrait d'une image une liste des contours détectés;
- **save** qui enregistre dans un fichier les contour détectés;

Si l'on considère que ces fonctions ont pour type :

$$\begin{aligned}\mathbf{thresh} &: \text{Image} \rightarrow \text{Image} \\ \mathbf{edge} &: \text{Image} \rightarrow [\text{Ligne}] \\ \mathbf{save} &: [\text{Ligne}] \rightarrow \emptyset\end{aligned}$$

où `Image` et `Ligne` représente les types de données encapsulant une image en niveaux de gris et les paramètres décrivant un segment au sein d'une image et où \emptyset désigne le type vide, l'application séquentielle initiale peut s'exprimer ainsi :

$$\mathcal{A}_s = \mathbf{Sequence} [\mathbf{thresh}, \mathbf{edge}, \mathbf{save}]$$

Un premier niveau de parallélisme peut être introduit en utilisant le squelette **Pipeline**. En effet, le traitement d'une image du flux étant indépendant du traitement de l'image précédente, il est possible de traiter simultanément les images i_t, i_{t+1} et i_{t+2} . On obtient alors :

$$\mathcal{A}_1 = \mathbf{Pipeline} [\mathbf{thresh}, \mathbf{edge}, \mathbf{save}]$$

Un deuxième niveau de parallélisme peut alors être appliqué à cette expression. En effet, l'étape de seuillage étant une opération iconique régulière, il est possible — via le squelette **SCM** — de paralléliser le traitement en découplant l'image initiale en bandes horizontales qui vont être seuillées indépendamment. Si l'on note :

$$\begin{aligned}\mathbf{scatter} &: \text{Image} \rightarrow [\text{Image}] \\ \mathbf{merge} &: [\text{Image}] \rightarrow \text{Image}\end{aligned}$$

Les fonctions séquentielles permettant le découpage et la fusion de sous-images, une nouvelle version parallèle de \mathcal{A}_s est exprimable en combinant les squelettes **Pipeline** et **SCM**.

$$\mathcal{A}_2 = \mathbf{Pipeline} [\mathbf{SCM} [\mathbf{scatter}, \mathbf{thresh}, \mathbf{merge}], \mathbf{edge}, \mathbf{save}]$$

5.2 Interface utilisateur

La bibliothèque QUAFF se propose de fournir une implantation C++ de ce modèle de programmation en fournissant une interface la plus proche possible de ce dernier. Pour ce faire, elle s'appuie sur :

- un jeu de squelettes supportant de manière transparente l'imbrication et plusieurs mécanismes d'optimisation;
- un mécanisme de gestion d'entrée/sortie qui permet d'intégrer la notion de flux au sein des applications;
- une méthode d'intégration des fonctions définies par l'utilisateur qui permet la réutilisation de code pré-existant.

L'originalité de QUAFF réside dans le fait que l'ensemble de ces mécanismes et des fonctionnalités associées s'appuie sur une implantation — décrite dans la section 5.3 — à base de métaprogrammes *templates* dont l'évaluation permet de générer un code efficace, très proche d'un code MPI écrit à la main.

5.2.1 Définition des tâches séquentielles

Afin de permettre la réutilisation de code existant, QUAFF fournit un mécanisme permettant à des fonctions simples et des foncteurs⁷ de s'intégrer de manière homogène au sein du code parallèle généré. Ce mécanisme repose sur l'utilisation d'un type *template* task qui prend en charge l'appel d'un foncteur ou d'une fonction depuis le code QUAFF. Ainsi, si l'on considère un foncteur C++ séquentiel comme celui présenté dans listing 5.1.

Listing 5.1 – Exemple de foncteur C++

```

1 struct Threshold
2 {
3     Image operator() ( const Image& in );
4 };

```

son intégration au sein d'une application QUAFF via l'utilisation de la structure task est présenté dans le listing 5.2.

⁷En C++, un foncteur est défini comme une classe exposant un opérateur () dans son interface et permettant donc d'utiliser ces instances comme de simples fonctions.

Listing 5.2 – Intégration d'un foncteur C++ au sein de QUAFF

```
1 typedef task<Threshold, Image, Image> thresh;
```

Le prototype de `task` comprend donc : le type du foncteur, son type d'entrée et son type de sortie. Dans le cas d'une fonction C++, la définition de la tâche associée n'est pas possible directement car le nom d'une fonction est une valeur et non un type. Elle passe donc par l'utilisation de la macro-définition `function` qui opère de manière similaire à `task` en prenant en compte cette spécificité⁸.

Listing 5.3 – Intégration d'une fonction C++ au sein de QUAFF

```
1 Image threshold( const Image& in );
2
3 typedef function(threshold, Image, Image) thresh;
```

Une fois ces types définis, les fonctions et foncteurs associés sont prêts à être utilisés au sein d'un squelette QUAFF.

5.2.2 Interfaces des squelettes

L'interface de QUAFF repose sur un constat simple : la définition d'une application sous la forme d'instances de squelettes algorithmiques est fondamentalement statique. QUAFF fournit donc des équivalents *templates* des squelettes exposés dans les sections 5.1.2 à 5.1.4 et des listes de fonctions qui permettent de décrire de telles applications.

5.2.2.1 Définition des listes de fonctions

Le modèle de programmation présenté au paragraphe précédent se base sur l'application de fonctions d'ordre supérieur sur des fonctions ou listes de fonctions définissant les applications à paralléliser. Les premiers outils fournis par QUAFF permettent de construire de telles listes. On distingue deux types d'arguments au sein des squelettes QUAFF :

- les arguments de type listes de fonctions comme celles utilisées par les squelettes comme **Pipeline**, **Sequence** ou **Pardo**. Au sein de QUAFF, ces

⁸Nous verrons dans la section 5.3 comment ces deux constructions sont liées.

fonctions sont représentées par des types (cf section 5.2.1) et ces listes sont construites via une structure *template stage* qui peut recevoir comme argument de un à seize types représentant des fonctions définies par l'utilisateur :

```
template<class F1, ..., class F16> class stage;
```

Si l'on considère une application \mathcal{A} définie comme :

$$\mathcal{A} = \textbf{Pipeline} [\textbf{task}_1, \textbf{task}_2, \textbf{task}_3]$$

La liste de fonctions $[\textbf{task}_1, \textbf{task}_2, \textbf{task}_3]$ est exprimée avec QUAFF par la définition de type suivante :

```
typedef stage<task_1,task_2,task_3> steps;
```

- les arguments issus de la réPLICATION d'une fonction. Les squelettes **Farm** et **SCM** utilisent dans leur définition une unique fonction qui est répliquée sur N processus esclaves. La structure `worker` permet de définir une telle fonction. Son prototype est :

```
template<class FUNC, size_t N> class worker;
```

Dans cette déclaration, le type `FUNC` représente la fonction à répliquer et `N` indique le nombre de réPLICATION. Par exemple, Un squelette **Farm** utilisant une fonction `foo` répliquée 10 fois utilisera en argument le type

```
typedef worker<foo,10> slaves;
```

L'ensemble de ces structures participent aussi à la vérification de types. Ainsi, au moment de sa définition la structure `stage` analyse les types d'entrée et de sortie de chacun de ses éléments et force, le cas échéant, le compilateur à émettre un message d'erreur indiquant les éléments erronés de la liste.

5.2.2.2 Syntaxe concrète des squelettes

Les squelettes fournis par QUAFF sont implantés en C++ via des structures *templates* dont les arguments décrivent la liste de fonctions utilisés pour leurs instantiation. Le listing 5.4 résume les prototypes C++ de ces structures.

Listing 5.4 – Classes définissant les squelettes QUAFF

```

1 template<class T, class V, class F> class select;
2 template<class STAGE> class sequence;
3
4 template<class STAGE> class pipeline;
5 template<class STAGE> class pardo;
6
7 template<class WORKER> class farm;
8 template<class S, class WORKER, class M> class scm;

```

L’instanciation d’un squelette via cette interface passe par l’instanciation d’une de ces classes *templates* en utilisant les structures stage pour les classes sequence, pipeline et pardo ou worker pour les classes farm, scm.

5.2.3 Définition des entrées/sorties

L’utilisation d’un flux de données au sein d’un code C++ peut se faire de plusieurs manières. On peut par exemple expliciter l’aspect temporel du flux en utilisant une boucle `for` qui énumère les éléments du flux ou bien utiliser des fichiers contenant le flux. D’autres approches, comme celle utilisé par la bibliothèque Muesli, utilisent des fonctions séquentielles spéciales : une **source** qui génère les éléments du flux d’entrées et un **puits** qui absorbe les éléments du flux de sortie. La solution retenue par QUAFF est similaire à celle de Muesli. Elle met en oeuvre une paire de fonctions dédiées à la génération des éléments du flux et à l’absorption du résultat de l’application sur le flux de données d’entrées. Formellement, on définit une source et un puits comme des fonctions dont le type d’arguments — respectivement le type de retour — est le type «vide» représenté au sein de QUAFF par le type `none_t`. Ainsi, le listing 5.5 présente le code du foncteur `LoadImage`.

Listing 5.5 – Définition du code d’une source

```

1 struct LoadImage
2 {
3     Image operator() () { return cam.getFrame(); }
4     Camera cam;
5 };

```

Son adaptation en vue d'une intégration au sein de QUAFF produit le code présenté dans le listing 5.6

Listing 5.6 – Adaptation du code d'une source

```

1 struct LoadImage
2 {
3     source_of<Image> operator() ( none_t )
4     {
5         if(cam.isOn()) return cam.getFrame();
6         else return Terminate<Image>();
7     }
8     Camera cam;
9 };

```

Les modifications incluent la modification du type de retour par le type *template* `source_of` et l'utilisation de `none_t`. Son code proprement dit consiste à tenter de récupérer une image depuis une caméra via la fonction `getFrame`. Si cette acquisition se révèle impossible — la camera est hors service par exemple —, `LoadImage` termine le flux de donnée en transmettant un marqueur de fin de flux via la fonction *template* `Terminate`.

La définition d'un puits se base sur le même principe. Considérons donc le foncteur `SaveImage` (listing 5.7).

Listing 5.7 – Définition du code d'un puits

```

1 struct SaveImage
2 {
3     SaveImage() : p("./image%04i.png", 0) {}
4     void operator() ( const Image& in ) { in.save(p++); }
5     Path p;
6 };

```

La sauvegarde de l'image se fait ici via l'interface de la classe `Image` qui utilise la classe `Path` qui encapsule la manipulation de chemin de fichier, chaque image reçue étant sauvé dans un fichier différent dont le nom est calculé par `Path`. Le listing 5.8 présente alors les changements nécessaires pour l'intégration au sein de QUAFF. On note l'utilisation du type `none_t` en lieu et place de `void` et l'ajout d'une directive `return idoine`.

Listing 5.8 – Adaptation du code d'un puits

```

1 struct SaveImage
2 {
3     SaveImage() : p("./image%04i.png", 0) {}
4     none_t operator()( const Image& in )
5     {
6         in.save( p++ );
7         return none_t();
8     }
9     Path p;
10 };

```

5.2.4 Définition d'une application QUAFF

Une fois les fonctions séquentielles définies et préparées à leur intégration dans une application QUAFF, il ne reste qu'à définir l'application elle-même en utilisant les diverses classes C++ représentant les squelettes fournis par QUAFF. Cette définition va passer par l'instanciation de divers types *templates* qui représentent les squelettes algorithmiques. Ainsi, si nous reprenons l'application \mathcal{A}_2 décrite précédemment :

$$\mathcal{A}_2 = \textbf{Pipeline} [\textbf{load}, \textbf{SCM} [\textbf{scatter}, \textbf{tresh}, \textbf{merge}], \textbf{edge}, \textbf{save}]$$

sa définition en C++ via l'interface de QUAFF est présentée sur le listing 5.9.

Listing 5.9 – Définition d'une application QUAFF

```

1 typedef task<LoadImage, none_t, Image> load;
2 typedef task<ScatterImg, Image, Image> scatter;
3 typedef task<Threshold, Image, Image> thresh;
4 typedef task<MergeImg, Image, Image> merge;
5 typedef task<Edge, Image, vector<Line>> edge;
6 typedef task<SaveLine, vector<Line>, none_t> save;
7
8 typedef scm<scatter, worker<thresh, 8>, merge> process;
9 typedef pipeline<stage<load, process, edge, save>> app;
10
11 application<app> detector;
12 detector.run();

```

Ce listing débute par la définition des tâches qui vont composer l’application finale (lignes 1–6) grâce à la classe *template task*. Les lignes 8 à 9 définissent les squelettes utilisé par $\mathcal{A}_{\textit{pipescm}}$, à savoir un squelette `scm` et un squelette `pipeline`. Enfin, l’application en elle même est définie à la ligne 11 par la classe `application` qui fournit un point d’entrée pour son exécution via la méthode `run`.

Le point important qui fonde le principe de l’interface de QUAFF est le fait que l’ensemble des définitions des tâches séquentielles, des squelettes et de leurs imbrications passent par de simples définitions de types, c’est à dire des constructions purement statiques résolues à la compilation. On note aussi que cette interface permet d’imbriquer naturellement des squelettes en conservant une écriture polymorphique.

5.3 Implantation

L’implantation de QUAFF est basée sur l’utilisation d’un système d’évaluation partielle d’une application parallèle exprimée sous forme de squelettes algorithmiques. Si l’on considère une telle application parallèle, il apparaît que la structure des squelettes qui la composent — c’est à dire la manière dont les squelettes sont imbriqués et instanciés — est connue au moment de la compilation. Les seuls éléments dynamiques de cette description sont les données du flux de données. Il paraît alors naturel de fournir un mécanisme capable de décrire de manière statique la structure d’une telle application et de permettre au compilateur d’analyser cette description afin de générer un code résiduel ne comprenant que des primitives de communications et des appels de fonctions séquentielles. La mise au point d’un tel système implique de répondre à deux types de besoins. Tout d’abord, il va être nécessaire de manipuler de manière statique des types C++ encapsulant les fonctions définies par l’utilisateur. Or, la représentation de fonctions sous la forme d’instances est un problème complexe à gérer dynamiquement. Il faut se tourner de nouveau vers une solution basée sur la métaprogrammation *template* et développer un ensemble de structures qui vont manipuler ces fonctions d’une manière très semblable à celle des langages fonctionnels et nous permettre ainsi d’écrire un équivalent C++ des fonctions d’ordre supérieur. L’autre point important est de fournir des mécanismes assurant une exécution performante de l’application finale. Ceci passe par un mécanisme de génération de code parallèle prenant en charge l’optimisation de l’expression formelle de l’application, la création de schémas de communication et une gestion efficace des transferts de données entre les processus.

5.3.1 Manipulation des listes de fonctions

La représentation et le stockage d'une liste de fonctions en C++ pose un problème intéressant. Une première approche repose sur une représentation dynamique via des pointeurs de fonctions. Cette approche est simple et naturelle mais se heurte à un problème d'hétérogénéité. En effet, en C et en C++, le type d'un pointeur de fonction dépend des arguments et du type de retour de la fonction pointée. Stocker des pointeurs de fonctions vers des fonctions exhibant des prototypes différents est alors impossible car il n'existe pas d'idiome naturel pour implanter des conteneurs hétérogènes. Une autre approche consiste à définir une hiérarchie de classes *templates* héritant toutes d'une classe abstraite représentant le concept général de fonction. Stocker de telles instances revient à gérer un tableau de pointeurs vers cette classe mère et laisser les mécanismes du polymorphisme de classe opérer. Cette approche règle le problème du stockage hétérogène et, grâce à divers systèmes mêlant *template* et fonctions virtuelles, permet de gérer des fonctions aux prototypes très différents. Une telle approche est par exemple utilisée par Kuchen au sein de Muesli. Dans ce type d'implantation, les squelettes sont alors représentés par des classes qui encapsulent les primitives de communications MPI nécessaires à l'expression du parallélisme et utilisent des conteneurs classiques pour conserver la liste des fonctions qui les composent. L'imbrication des squelettes est alors possible en faisant dériver les classes de squelettes de la même classe mère que les fonctions séquentielles. Cette solution reste néanmoins peu satisfaisante en terme de performance. En effet, on note que les applications écrites en utilisant une telle interface sont en moyenne 1.5 à 2 fois moins performantes que leurs équivalents écrits directement en MPI [119], ce qui peut pour certains types d'applications — comme par exemple des applications de vision temps réel — être rédhibitoire. Ces pertes proviennent en grande partie du surcoût dû à la résolution des appels de fonctions virtuelles. Considérons par exemple une classe C++ exposant dans son interface une méthode classique et une méthode virtuelle (listing 5.10).

Listing 5.10 – Exemple de classe exposant une méthode virtuelle

```

1 struct FOO
2 {
3     void             f ();
4     virtual void    g ();
5     int              mData;
6 }
```

Une instance de cette classe (figure 5.7) est alors composée de :

- Une zone dédiée aux données membres;
- Une zone contenant un pointeur vers le constructeur et la méthode `f`;
- Une zone contenant un pointeur vers la table des fonctions virtuelles ou *vtable*.

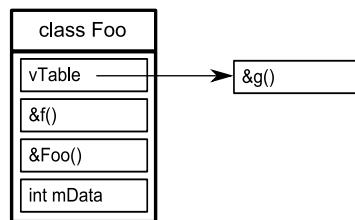


FIG. 5.7 – Structure d'une classe C++ : membres, méthodes et *vtable*

Lorsque un appel à `Foo::f` est résolu, le compilateur génère un simple appel via le pointeur de fonction contenu dans la classe. Lorsque un appel à `Foo::g` est résolu, le compilateur doit tout d'abord accéder au contenu de la *vtable*, récupérer le pointeur correspondant à `Foo::g` et résoudre l'appel. Ce mécanisme génère des accès mémoire supplémentaires qui ralentissent d'autant l'exécution de cette méthode. En outre, peu de compilateurs sont capables d'effectuer des optimisations de code autour d'un appel de fonction virtuelle, rendant inefficaces les stratégies de placement des instructions ou des données dans le cache, le déroulage des boucles et les mécanismes d'*inlining*. La politique d'utilisation des fonctions virtuelles dans un code C++ est donc d'utiliser ce mécanisme uniquement si la quantité de code à exécuter au sein de la fonction est importante et si la fonction n'est pas appelée au sein d'une boucle critique. Malheureusement, dans le cas de codes de calcul parallèles, les fonctions qui s'implanterait de manière élégante en tant que fonctions virtuelles sont justement des fonctions critiques. Dans notre cas précis, lorsqu'une classe de squelette doit parcourir l'ensemble de ses éléments, elle doit procéder tout d'abord au déréférencement du pointeur vers chacune des fonctions qui la composent puis résoudre un appel de fonction virtuelle. Lors de l'exécution de l'application, ces temps d'accès s'accumulent et deviennent prohibitifs. La solution que nous proposons repose sur deux points : la définition d'une classe *template* permettant d'encapsuler de manière homogène une fonction quelconque et la définition d'une classe *template* permettant de gérer l'équivalent d'un conteneur de type statique.

5.3.1.1 Adaptateur fonction/foncteur

Une des qualités que l'on peut attendre d'une bibliothèque basée sur les squelettes algorithmiques est de permettre la réutilisation d'un large panel de code existant [44]. Pour ce faire, QUAFF propose un système permettant d'utiliser à la fois des fonctions et des foncteurs⁹ en posant un minimum de contraintes sur leur prototypes. L'idée principale est de fournir une classe *template* qui va permettre de réunir les informations nécessaires à l'exécution de ces fonctions et de les encapsuler dans une classe au prototype pré-défini et compatible avec l'interface des squelettes. Au sein de QUAFF, ce sont les constructions `task` et `function` définis au paragraphe 5.2.1 qui remplissent ce rôle. Nous savons que leurs arguments *templates* permettent de spécifier le type du foncteur, son type d'argument et son type de retour. À partir de ces données, la construction `task` évalue les types nécessaires au fonctionnement interne des squelettes¹⁰. La construction `function` quant à elle permet de transformer une fonction libre en un type utilisable par QUAFF. En effet, il n'est pas possible en C++ d'instancier un objet de type «poiniteur de fonction». L'écriture correcte nécessiterait de créer un foncteur dont l'appel seraient transmis à la fonction libre qu'il encapsule. Or, il n'est pas envisageable de requérir de l'utilisateur qu'il fournit ces foncteurs additionnels. La macro-définition `function` permet de contourner ce problème en masquant la création du foncteur intermédiaire (listing 5.11).

Listing 5.11 – Définition de la macro `function`

```

1 #define function(F, IN, OUT)          \
2 struct _QUAFF_INNER_ ## F           \
3 {                                     \
4     _QUAFF_INNER_ ## F () {}         \
5     OUT operator()( const IN& i )   \
6     {                                 \
7         return F(i);                 \
8     }                                 \
9 } _QUAFF_ ## F;                     \
10                                         \
11 typedef task<_QUAFF_ ## F, IN, OUT>

```

Grâce à cette écriture, l'intégration de fonctions et de foncteurs au sein de QUAFF se fait de manière homogène (listing 5.12)

⁹Au sens C++ du terme.

¹⁰Fonctionnement que nous détaillerons plus avant au paragraphe 5.3.2

Listing 5.12 – Utilisation de fonction et task

```

1 struct Functor
2 {
3     int operator() (const int& in)
4 };
5
6 double func( const string& in );
7
8 typedef task<Functor,int,int> myFunctor;
9 typedef function(func,string, double) myFunction;

```

5.3.1.2 Conteneur statique de type

La définition d'un conteneur de types est relativement complexe au premier abord. En effet, les types C++ n'étant pas des objets de premier ordre¹¹, il n'est pas possible d'utiliser les outils standards du langage pour définir une telle structure de donnée. Une solution classique au problème du conteneur de type repose sur l'utilisation d'un idiomme proposé par Alexei Alexandrescu [10] — la liste statique de types — qui consiste à définir un conteneur dont l'implantation est très semblable à celle d'une liste dans un langage fonctionnel comme ML. Une telle liste est constituée d'une «tête» contenant un unique élément et d'une «queue» contenant soit une liste soit l'élément de terminaison. Le listing 5.13 présente la transcription de cette définition sous la forme d'une structure *template*.

Listing 5.13 – Implantation de typelist

```

1 template<class HEAD, class TAIL = null_t> struct typelist
2 {
3     typedef HEAD head_t;
4     typedef TAIL tail_t;
5 };

```

La structure typelist définie à la ligne 3 admet comme paramètres *templates* le type de son élément de tête (HEAD) et de son élément de queue (TAIL). Ce dernier prend par défaut une valeur signifiant la terminaison de la liste. Le listing 5.14 présente la définition d'une liste contenant l'ensemble des types entiers¹².

¹¹C'est à dire qu'il n'est pas possible de créer une variable contenant un type.

¹²à savoir char, short et int.

Listing 5.14 – Création d'une liste de types

```
1 typelist<char, typelist<short, typelist<int>>>
```

La structure `typelist` permet donc, de part son caractère récursif, de stocker un nombre illimité de types. mais la syntaxe présentée dans le listing 5.14 devient néanmoins rapidement lourde pour la définition de listes de grande taille. Pour permettre de définir de manière plus simple une telle liste de types, la métा-fonction `make_list` est proposée. Cette métा-fonction `make_list` définit de manière récursive une `typelist` à partir d'une liste linéaire d'arguments *templates* (cf listing 5.15). Contrairement à `typelist`, il est nécessaire de spécifier à l'avance un nombre maximal d'argument (en général une dizaine).

Listing 5.15 – Définition de `make_list`

```
1 template<class T1=null_t, class T2=null_t, class T3=null_t
2           , class T4=null_t, class T5=null_t, class T6=null_t
3           , class T7=null_t, class T8=null_t, class T9=null_t
4           >
5 struct make_list
6 {
7     typedef typename make_list<T2, T3, T4, T5,
8                           T6, T7, T8, T9>::type_t tail_t;
9     typedef typelist<T1, tail_t> type_t;
10 };
11
12 template<> struct make_list<null_t>
13 {
14     typedef null_t type_t;
15 };
```

À titre d'exemple, le listing 5.16 montre comment utiliser `make_list` pour définir une liste statique identique à celle du listing 5.14.

Listing 5.16 – Utilisation de `make_list`

```
1 typedef make_list<char, short, int>::type_t integers_t;
```

Il est alors possible de définir des métaprogrammes récursifs qui vont manipu-

ler ces structures afin de fournir l'équivalent statique des opérations classiques¹³ sur les conteneurs : ajout et retrait d'élément, recherche d'un élément, accès aléatoire à un élément, concaténation de listes. La mise en œuvre de cet idiome au sein de QUAFF se fait au sein des types `stage` et `worker` qui sont des structures utilisant `make_list` afin de générer la liste des fonctions qui leur est associée. Ainsi, `stage` est une simple macro-définition remplaçant l'appel à `make_list` et `worker` utilise une structure récursive pour générer la liste statique qu'elle définit (listing 5.17).

Listing 5.17 – Définition de `worker`

```

1 template<class T, size_t N> struct worker
2 {
3     typedef typename worker<T,N-1>::type_t base_t;
4     typedef typelist<T,base_t> type_t;
5 };
6
7 template<class T> struct worker<T,0>
8 {
9     typedef null_t type_t;
10 };

```

Si l'on considère la déclaration d'une application QUAFF simpliste (listing 5.18) composé d'un squelette **Farm** utilisant 4 processus esclaves pour exécuter un **Pipeline** de deux étapes :

Listing 5.18 – Exemple d'application QUAFF

```

1 typedef task<task1,none_t,int> task_1;
2 typedef task<task2,none_t,int> task_2;
3
4 typedef stage<task_1,task_2> steps;
5 typedef pipeline<steps> pipe;
6 typedef farm<worker<pipe,4> app;

```

La type `app` ainsi défini est alors équivalent à (listing 5.19) :

À partir de cette représentation, nous allons voir comment le générateur de code de QUAFF va forcer le compilateur à écrire un code MPI optimal.

¹³Pour référence, les codes sources de l'ensemble de ces opérations sont donnés en annexe IV.

Listing 5.19 – Liste de type associée à `farm<worker<pipe, 4>`

```

1 farm <
2   typelist<pipeline<typelist<task_1, typelist<task_2>>>,
3   typelist<pipeline<typelist<task_1, typelist<task_2>>>,
4   typelist<pipeline<typelist<task_1, typelist<task_2>>>,
5   typelist<pipeline<typelist<task_1, typelist<task_2>>>
6   >

```

5.3.2 Mise en œuvre du parallélisme

Une fois les listes de fonctions définies et les squelettes instanciés, il faut permettre au compilateur d’interpréter ces listes comme un modèle pour la génération d’un code parallèle MPI afin de produire un code exécutable prenant en charge le placement des processus et la gestion des communications. Il y a donc ici trois étapes à exécuter les unes après les autres. La première étape consiste à reformuler le schéma à base de squelettes définissant une application afin de détecter les imbrications illicites, de simplifier certaines imbrications de squelettes et de préparer le code de la classe `application` à être instancié. La seconde étape effectue le placement des processus sur les processeurs mis à disposition par la machine parallèle sous-jacente. Cette étape utilise des méta-fonctions qui vont générer un code dont l’exécution va permettre le placement effectif des processus. Enfin, la dernière étape prend en charge l’exécution des fonctions ainsi définies et leurs communications.

5.3.2.1 Optimisation de l’application

L’optimisation d’une application décrite par une imbrications de squelettes algorithmiques consiste à repérer des schémas redondants au sein de cette définition et de les éliminer tout en conservant la sémantique fonctionnelle de l’application. Ainsi, l’application effectivement instanciée va nécessiter l’utilisation d’un nombre réduit de processus et, de fait, limiter les étapes de communications. On peut en profiter également pour détecter des constructions illicites.

Dans ses travaux, Aldinucci introduit un formalisme [5, 4] qui permet de démontrer que tous les squelettes algorithmiques sont exprimables sous une forme dite normale. Il montre alors qu’optimiser une application à base de squelettes revient à appliquer une série de transformations sur celle-ci. Pour cela, Aldinucci définit la **Frange** d’une application comme étant la liste ordonnée de l’ensemble

des tâches séquentielles contenues dans cette application. Cette fonction permet alors de définir la forme normale $\overline{\mathcal{A}}$ d'une application \mathcal{A} comme :

$$\overline{\mathcal{A}} = \mathbf{Farm}(\mathbf{Sequence}(\mathbf{Frange}(\mathcal{A})))$$

Il en découle que toute application utilisant des squelettes peut être reformulée sous la forme d'un squelette **Farm** exécutant une **Sequence** contenant l'ensemble ordonnées des tâches séquentielles de l'application initiale. Aldinucci et Dane-lutto démontrent [6] en outre que cette réécriture améliore le temps de mise à disposition — c'est à dire le temps nécessaire pour le résultat du traitement d'un élément du flux de données soit effectué — et le temps d'exécution total — c'est à dire le temps compris entre l'entrée du premier élément du flux de données et la sortie du résultat correspondant au traitement du dernier élément du flux. En outre, l'utilisation de la forme normale permet de réduire le nombre de processus utilisés et augmente le degré de parallélisme réellement exploité.

Néanmoins, au sein de QUAFF, la transformation d'une application en sa forme normale n'est pas entièrement possible et ce pour deux raisons. Tout d'abord, les squelettes comme **SCM** ou **Pardo** ne font pas partie des squelettes considérés dans ces travaux. Ensuite, transformer une application en sa forme normale requiert de connaître le nombre de processeurs disponibles. Or, dans notre cas, cette transformation est effectuée lors de la compilation à un instant où ce nombre n'est pas connu. Nous avons choisi néanmoins d'utiliser le formalisme et les règles édictées par Aldinucci afin de gérer de manière transparente un sous-ensemble d'optimisations locales. Ces règles s'expriment formellement ainsi :

- Les compositions de **Pipeline** ou **Sequence** se simplifient en une unique instance de squelette :

$$\begin{aligned} \mathbf{Pipeline}[f_1, \dots, \mathbf{Pipeline}[f_{i+1}, \dots, f_j], \dots, f_n] &\equiv \mathbf{Pipeline}[f_1, \dots, f_n] \\ \mathbf{Sequence}[f_1, \dots, \mathbf{Sequence}[f_{i+1}, \dots, f_j], \dots, f_n] &\equiv \mathbf{Sequence}[f_1, \dots, f_n] \end{aligned}$$

- Les squelettes **Pipeline** et **Farm** sont interchangeables, la forme **Farm** \circ **Pipeline** étant la plus efficace.

$$\mathbf{Pipeline}[f_1, \dots, \mathbf{Farm}[g], \dots, f_n] \equiv \mathbf{Farm}[\mathbf{Pipeline}[f_1, \dots, g, \dots, f_n]]$$

De par la nature récursive des listes de types et des métas-fonctions qui les manipulent, la définition d'un équivalent *template* de ces règles est relativement aisée. Ainsi, chaque classe de squelette expose deux types : `optim_t` et `optim_arg_t`

qui représentent respectivement la forme optimisée du squelette et la liste de type optimisée de ce même squelette. Le listing 5.20 présente ces types pour la classe `pipeline` :

Listing 5.20 – Aide à l’optimisation de Pipeline

```

1 template<class STG> class pipeline
2 {
3     public:
4     typedef typename STG::type_t arg_t;
5     typedef typename opt_pipe<arg_t>::type_t optim_arg_t;
6     typedef pipeline<optim_arg_t> optim_t;
7     // ...
8 }
```

Ces types sont aussi définis au sein de la classe `task` (listing 5.21).

Listing 5.21 – Aide à l’optimisation de Pipeline

```

1 template<class FUNC, class I, class O> class task
2 {
3     public:
4     typedef task<FUNC, I, O> optim_t;
5     typedef task<FUNC, I, O> optim_arg_t;
6     // ...
7 }
```

Chaque squelette est ensuite associé à une classe — ici `opt_pipe` — qui prend en charge l’optimisation interne de sa liste d’arguments. Le listing 5.22 illustre le principe général de ces méta-programmes en présentant le code de `opt_pipe`. La structure récursive de `opt_pipe` est évidente. Après avoir défini le cas général de son application (lignes 3–6), nous définissons successivement un cas particulier d’optimisation (lignes 8–14) qui prend en charge l’élimination des pipeline internes et un cas terminal (lignes 16–19) qui permet de stopper la récursion. Des structures semblables sont définies pour chaque règle d’optimisation. Rappelons encore une fois que l’ensemble de ces traitements sont effectués par le compilateur et ne grève en rien les performances finales de l’application.

Listing 5.22 – Méta-fonction d’optimisation des imbriques de Pipeline

```

1 template<class T> struct opt_pipe {};
2
3 template<class H, class T>
4 struct opt_pipe<typelist<H, T>>
5 {
6     typedef typename H::optim_arg_t           h_t;
7     typedef typelist<h_t, opt_pipe<T>::type_t> type_t;
8 };
9
10 template<class P, class R>
11 struct optimize<typelist<pipeline<P>, R>>
12 {
13     typedef typename pipeline<P>::optim_arg_t h_t;
14     typedef typename opt_pipe<R>::type_t      t_t;
15     typedef typename append<h_t, t_t>::type_t type_t;
16 };
17
18 template<> struct optim_pipe<null_t>
19 {
20     typedef null_t type_t;
21 };

```

5.3.2.2 Instanciation d’une application QUAFF

Une fois que la liste de types décrivant l’application est optimisée, il nous reste donc à définir un moyen d’instancier les types qu’elle contient. Pour cela, en s’appuyant sur l’aspect récursif de typelist, on définit alors une structure tuple (listing 5.23).

Listing 5.23 – Définition de la classe tuple

```

1 template<class TL>
2 struct tuple : public tuple<typename TL::tail_t>
3 {
4     typename TL::head_t mValue;
5 };
6
7 template<> struct tuple<null_t> {};

```

Cette classe parcourt les types contenus dans la liste de types qui lui est passée en argument *template* et instancie le type de tête de cette dernière. En outre, il hérite de l’instance de `tuple` basée sur la queue de la liste. Ainsi, la déclaration :

```
typedef tuple< make_list<Task1, Task2, Task3>::type_t > stages_t;
```

génère une classe dont la structure est celle présentée sur la figure 5.8

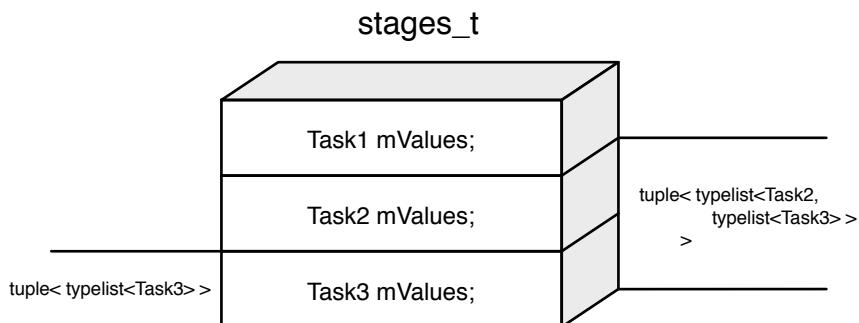


FIG. 5.8 – Structure mémorielle d’un `tuple`

Étant constitué d’instances de types divers, `tuple` se comporte comme un conteneur hétérogène et permet de s’affranchir de la définition d’une interface polymorphe sous forme d’une classe exhibant une ou plusieurs méthodes virtuelles et *de facto* va contribuer à réduire les pertes de performances liées à ce mécanisme. La phase finale de la compilation d’une application QUAFF passe par la création d’un tel `tuple` contenant l’ensemble des informations nécessaires à l’initialisation des squelettes et fonctions composant cette application. Chaque squelette contient en effet un membre de type `tuple` qui va générer le code nécessaire à son exécution. Par exemple, au sein de la classe `pipeline`, la liste de types passée en argument est optimisée puis le type *template* `tuple` est instancié (listing 5.24). Une fois instancié, `pipeline` contient une instance de chacune des fonctions et de chacun des squelettes imbriqués en son sein et est prêt à être exécuté. Chacun des squelettes supporté par QUAFF se comporte d’une manière similaire, le nombre de `tuple` instanciés étant variable.

Listing 5.24 – Instanciation d'un pipeline

```

1 template<class STAGES> class pipeline
2 {
3     public :
4     // ...
5     typedef typename opt_pipe<arg_t>::type_t optim_arg_t;
6     typedef tuple< optim_arg_t > stages_t;
7
8     // ...
9
10    private:
11    stages_t mStages;
12}

```

5.3.2.3 Placement des processus

L'exécution d'une application QUAFF débute par l'appel au constructeur de la classe *application*. Cet appel va générer des appels aux constructeurs de chaque fonction et squelette contenus dans les diverses instances de la classe *template tuple* contenus dans chaque squelettes. La première tâche effectuée par ces constructeurs est de procéder au placement de leurs fonctions sur les processeurs disponibles. Le problème du placement des tâches sur un ensemble de processeurs peut être résolu par deux approches :

- **Une approche basée sur les groupes de processeurs MPI** [92]. Au moment de son instantiation, chaque squelette s'approprie un sous-ensemble des processus disponibles en utilisant un algorithme spécifique à sa sémantique. Ces sous-ensembles sont alors utilisés pour construire l'instance de MPI_group et pour récupérer le communicateur MPI associé. A l'exécution, chaque élément d'un graphe de tâche possède son propre communicateur dans lequel il peut effectuer les communications nécessaires. Les communications entre squelettes sont ensuite gérées au sein d'inter-communicateurs qui relient entre eux les processeurs «racines» du communicateur de chaque squelette.
- **Une approche basée sur le placement linéaire des tâches** [118, 119]. Dans cette approche, chaque tâche est définie par le rang du processus qui

l'exécutera et par le rang des processus avec lesquels elle devra communiquer. Lorsque qu'une tâche ou un squelette est instancié, le rang de ses processus est calculé en utilisant un algorithme dépendant du squelette. Chaque squelette applique enfin une dernière étape qui permet de relier entre eux des squelettes potentiellement imbriqués. Lors de l'exécution de l'application, l'ensemble des communications se déroule dans MPI_COMM_WORLD, le communicateur principal de MPI.

Même si les deux approches conduisent à des résultats comparables en termes de performances, notre choix fut d'utiliser l'approche linéaire car elle se plie mieux au modèle d'instanciation des tâches que nous avons mis en place. Pour cela, il faut revenir à l'expression de chaque squelette sous la forme d'un graphe de processus communiquant et définir, pour chaque squelette, un algorithme permettant d'affecter à chacun de ces processus le rang qu'il convient. Pour cela, on décrit le placement d'une fonction ou d'un squelette par un triplet de valeurs :

- ID, qui représente le rang du processus sur lequel la fonction va s'exécuter;
- IN, qui représente le rang du processus qui va transmettre à la fonction ses données d'entrées;
- OUT, qui représente le rang du processus qui va recevoir la valeur de retour de la fonction considérée.

Le placement des processus va donc consister à fixer ces valeurs pour chaque fonction et squelette constituant l'application. On définit alors un certain nombre de fonctions qui vont effectuer le placement respectivement des processus d'une application complète, d'une fonction encapsulée dans une classe task ou d'un squelette quelconque.

- Pour les **applications**, cet algorithme consiste à demander le placement du squelette de plus haut-niveau contenu dans l'instance de l'application à partir du processus de rang 0.

```

1  Placer( application A )
2    Placer( A.squelette(), 0 );
3  Fin .

```

- Pour les **fonctions séquentielles** cet algorithme se réduit à sa plus simple expression. Il consiste à mettre à jour la valeur de l'identifiant de la tâche,

de son prédecesseur et de son successeur via la valeur du processus courant.

```

1  Placer( task  $T$ , int r )
2     $T$ .ID = r;
3     $T$ .IN = r;
4     $T$ .OUT = r;
5    Retourner r;
6  Fin.
```

- Pour le squelette **Pipeline**, le placement des fonctions qui le composent se décompose en deux parties La première partie consiste à parcourir les fonctions définissant le **Pipeline** et à les placer sur des processus de rang contigus. Une fois les appels récursifs terminés, on établit les communications entre chaque étage du **Pipeline** en affectant le rang des processus $j - 1$ et $j + 1$ en entrée et sortie du processus j .

```

1  Placer( pipeline  $\mathcal{P}$ , int r )
2     $\mathcal{P}$ .ID = r;
3    Retourner PlacerPipeline(  $\mathcal{P}$ .stage(), r, 0 );
4  Fin.
5
6  PlacerPipeline( [function]  $\mathcal{F}$ , int r, int i )
7    n = taille( $\mathcal{F}$ );
8    Si i < n Alors
9      id = Placer(  $\mathcal{F}$ [i], r );
10     id = PlacerPipeline(  $\mathcal{F}$ , id+1, i+1 );
11     Si i != 0 Alors  $\mathcal{F}$ [i].IN =  $\mathcal{F}$ [i-1].ID;
12     Si i != n Alors  $\mathcal{F}$ [i].OUT =  $\mathcal{F}$ [i+1].ID;
13     Retourner id;
14   Fin Si.
15   Retourner r;
16 Fin.
```

- Pour les squelettes **Farm** et **SCM**, l'algorithme consiste à placer le processus d'émission/réception puis à parcourir la liste des esclaves. Ensuite, une phase de correction des liens vient modifier les entrées/sorties des esclaves afin de pointer vers le processus de distribution/transfert. Dans le même temps, la liste des rangs des processus utilisés comme esclaves est construite.

```

1  Placer( farm  $\mathcal{F}$ , int r )
2     $\mathcal{F}.\text{ID} = r;$ 
3    Retourner PlacerFarm(  $\mathcal{F}.\text{worker}()$ , r, 0,  $\mathcal{F}.\text{workerID}$ );
4    Fin.
5
6  PlacerFarm( [task]  $\mathcal{S}$ , int r, int i, [int] workers)
7    n = taille( $\mathcal{S}$ );
8    Si i < n Alors
9      workers[i] = r;
10     id = Placer(  $\mathcal{S}[i]$ , r );
11     id = PlacerFarm(  $\mathcal{S}$ , id+1, i+1 );
12     Si i != 0 Alors  $\mathcal{F}[i].\text{IN} = r$ ;
13     Si i != n Alors  $\mathcal{F}[i].\text{OUT} = r$ ;
14     Retourner id;
15   Fin Si.
16   Retourner r;
17 Fin.

```

La seule différence entre **Farm** et **SCM** est que pour **SCM**, la fonction **Placer** renvoie le rang r au lieu de renvoyer le rang du prochain processus.

```

1  Placer( in : scm  $\mathcal{S}$ , int r )
2     $\mathcal{S}.\text{ID} = r;$ 
3    PlacerFarm(  $\mathcal{S}.\text{worker}()$ , r, 0,  $\mathcal{S}.\text{workerID}$ );
4    Retourner r;
5    Fin.

```

- Les squelettes **Pardo**, **Sequence** et **Select** procèdent très simplement. Après avoir récursivement parcouru leurs sous-éléments, ils procèdent à la correction des liens entre ces derniers en réinitialisant leur prédécesseur et leur successeur afin de n'effectuer aucune communication.

Considérons par exemple une application \mathcal{A} définie comme :

$$\mathcal{A} = \mathbf{Pipeline} [f_1, [\mathbf{Farm}_2 g], f_2]$$

Son expression au sein de QUAFF est donnée par le type :

pipeline< stage< F1, farm< worker<G,2>,F2 >

Le placement des tâches associées à cette définition est alors donné sur la figure 5.9. Dans ce schéma, nous représentons chaque fonction et squelette par un bloc contenant le nom de la fonction ou du squelette et un triplet de nombres représentant respectivement le rang du processus associé, le rang du processus d'entrée et le rang du processus de sortie. À chaque étape de l'algorithme, nous indiquons en rouge le processus en train d'être mis à jour et en bleu, les processus mis à jour. Le schéma global se lit de haut en bas au fur et à mesure de la progression des appels récursifs. À la fin de chaque étape de placement d'un squelette, nous indiquons en vert les rangs des processus d'entrée/sortie remis à jour.

L'ensemble de ces algorithmes sont implantés au sein de QUAFF par un ensemble de méta-programmes récursifs qui parcourrent les éléments des listes de types de chaque squelette composant une application et génère le code correspondant au placement adéquat. Le listing 5.25 illustre par exemple le code du méta-programme effectuant le placement des processus d'un **Pipeline**¹⁴.

Listing 5.25 – Méta-programme de placement pour **Pipeline**

```

1 template<class S>
2 int pipeline<S>::initTopology( int id )
3 {
4     static const N = length<S>::Value;
5     return pipe_topology<STG,N,0>::Init( mStages, id );
6 }
7
8 template<class S, size_t L, size_t I>
9 struct pipe_topology
10 {
11     static int Init( S& s, int id )
12     {
13         id = field<I>(s).initTopology(id);
14         id = pipe_topology<S,L,I+1>::Init(s, id++);
15         field<I>(s).Entrance(field<I-1>(s).ID());
16         field<I>(s).Exit(field<I+1>(s).ID());
17         return id;
18     }
19 }
```

¹⁴Afin de ne pas surcharger le manuscrit, seul le cas général de ce méta-programme est présenté. Le lecteur pourra trivialement déduire l'écriture des cas terminaux (pour $L = I$ et $I = 0$)

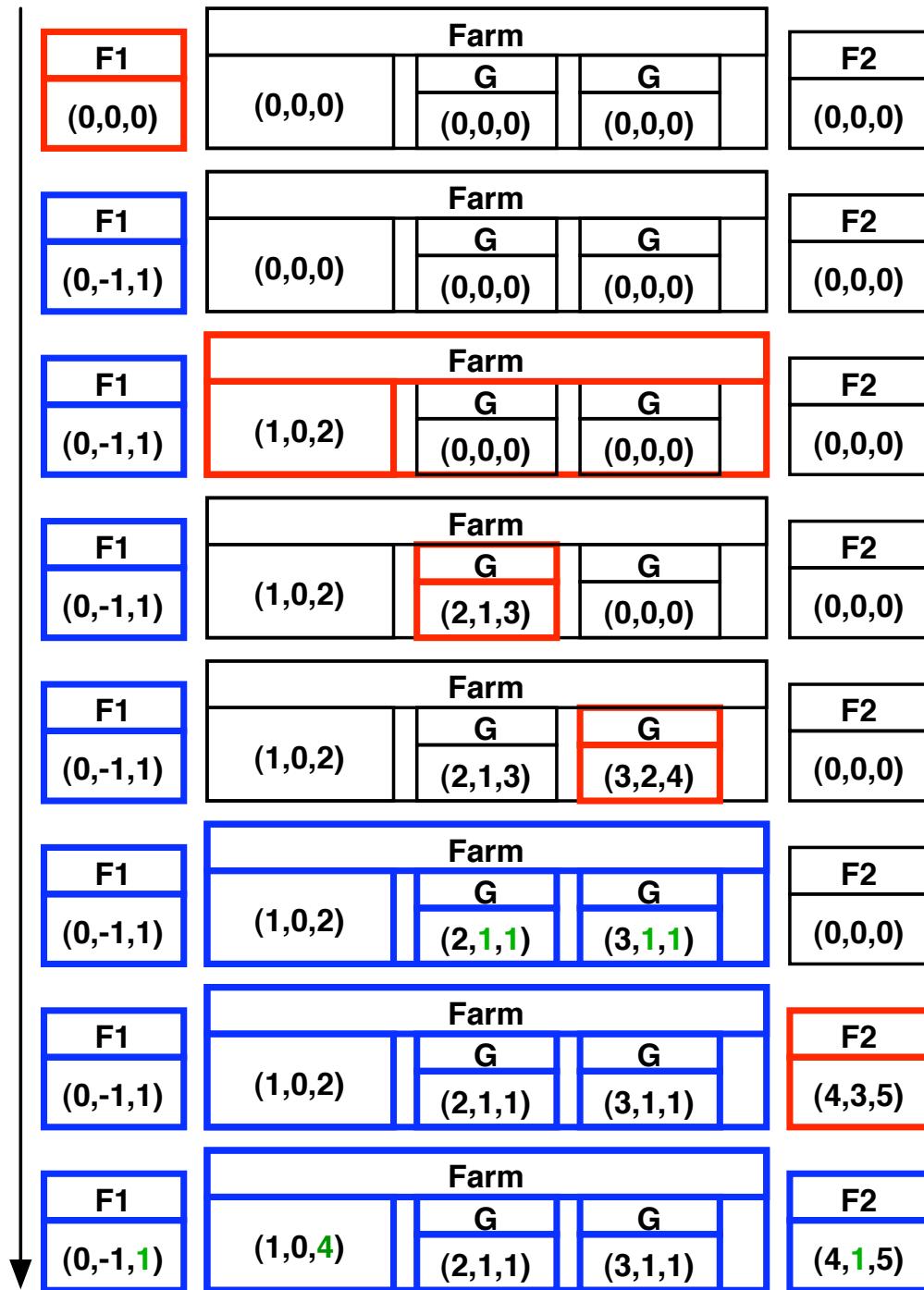


FIG. 5.9 – Déroulement de l'étape de placement des tâches

5.3.2.4 Exécution des fonctions utilisateurs

Une fois l’application construite, son exécution proprement dite est effectuée par l’appel de la méthode `run` de la classe `application`. Cette dernière appelle à son tour la méthode `run` de son squelette de plus haut-niveau. Dans ce squelette, chaque élément est alors exécuté par la même méthode, selon un schéma défini par la sémantique opérationnelle du squelette. Toutes les fonctions définies par l’utilisateur encapsulées par la classe `task` et tous les squelettes fournis par QUAFF exposent une telle méthode `run`, assurant ainsi que la requête d’exécution est transmise à tous les éléments de l’application. Par exemple, la classe `task` qui encapsule les fonctions utilisateurs possède une méthode `run` qui prend en charge l’attente d’une communication, le traitement de la donnée reçue et sa transmission. Chacune de ces étapes est exécuté jusqu’à ce qu’un élément indiquant la fin du flux soit reçue. Les primitives de communications MPI sont ensuite encapsulées dans une classe `Stream` qui prend en charge les cas particuliers des sources et des puits. La forme générale de la méthode `run` est donc :

Listing 5.26 – Méthode `run` de la classe `task`

```

1 void task<F, I, O>::run()
2 {
3     if(QUAFF::Rank() == myRank)
4     {
5         do { Stream::Receive(mInput, mEntrance);
6             mOutput = mFunction(mInput);
7             Stream::Send(mOutput, mExit);
8         } while(mOutput.isValid());
9         Terminate();
10    }
11 }
```

Cette méthode est évidemment très proche de celle utilisée dans l’implantation à base de méthodes virtuelles. La principale différence réside dans le fait que les appels effectués ici ne nécessitent pas la résolution d’un appel de méthode virtuelle. En effet, étant validés statiquement, il suffit que l’interface de chacune de ces classes (`task` et squelettes) expose une méthode `run`. Par un parcours récursif de chaque élément des classes tuple composant chaque squelette, l’appel de `run` est propagé à l’ensemble des fonctions de l’application. Le point important de cette exécution est la manière dont le flux de données est transmis de processus à processus en fonction de la sémantique opérationnelle de chaque squelette.

Ainsi, la méthode `run` de chaque squelette consiste à exécuter les méthodes `run` de chaque fonction interne puis à procéder aux éventuelles tâches annexes (comme par exemple gérer un processus de collecte/distribution dans le cas de **Farm**). Aucun test sur les rangs des fonctions n'est effectué à ce niveau. Si une tâche est exécutée sur un processus incorrect, le test au sein de la méthode `run` de `task` provoque son arrêt immédiat.

L'ensemble de ces méthodes sont déroulées en place lors de la compilation ce qui génère un code très compact et très proche du code équivalent MPI. Ainsi, si l'on prend l'exemple d'un **Pipeline** contenant trois fonctions, le code généré est équivalent à celui présenté dans le listing 5.27.

Listing 5.27 – Déroulage de la méthode `run` pour un squelette pipeline

```
1  if (QUAFF::Rank () == 0 )
2  {
3      do { mOutput = Task1 ();
4          Stream::Send(mOutput,1);
5      } while(mOutput.isValid());
6      Terminate ();
7  }
8
9  if (QUAFF::Rank () == 1 )
10 {
11     do { Stream::Receive(mInput,0)
12         mOutput = Task2(mInput);
13         Stream::Send(mOutput,2);
14     } while(mOutput.isValid());
15     Terminate ();
16 }
17
18 if (QUAFF::Rank () == 2 )
19 {
20     do { Stream::Receive(mInput,1)
21         mOutput = Task2(mInput);
22     } while(mOutput.isValid());
23     Terminate ();
24 }
```

5.3.3 Gestion des communications

Le dernier point à considérer au sein de l'implantation de QUAFF est la gestion des communications. Si le transfert de données dites atomiques (comme des entiers ou des nombres réels) peut se faire de manière native, la prise en compte des types définis par l'utilisateur est plus complexe. Deux situations se posent en pratique :

- le type défini par l'utilisateur peut être décrit comme un agrégat statique de types atomiques. C'est le cas par exemple des structures de données simples comme une structure encapsulant un triplet de coordonnées ou un tableau de taille fixe. Ces structures peuvent être intégrées au sein de MPI via l'utilisation de primitives comme `MPI_Type_contiguous` ou `MPI_Type_vector`.
- le type défini par l'utilisateur nécessite un protocole de transfert *ad hoc*. Par exemple, l'envoi et la réception via MPI d'un tableau de taille variable nécessite de définir une fonction qui, dans un premier temps, va transmettre la taille effective du tableau. Ensuite, du côté récepteur, une zone mémoire de taille adéquate est allouée. Enfin, les éléments du tableau sont transmis.

Le problème est alors de proposer une solution homogène pour gérer les transferts de données atomiques, de données composites et des données nécessitant un protocole dédié. L'idée proposée par QUAFF est d'encapsuler les données transmises entre chaque processus communicant dans une classe servant de passerelle¹⁵ entre l'interface homogène attendue par les squelettes et les fonctions et l'interface concrètement fournie par les éléments à transmettre. Cette classe, nommée `data` est une classe *template* dont l'interface est composée de trois parties :

- une série de méthodes permettant la construction, la copie et l'affectation d'une donnée de type `T` à une instance de `data<T>` afin de rendre transparent le passage d'argument de type `data<T>` en lieu et place d'arguments de type `T`. La classe `data<T>` stocke en son sein une référence vers la donnée à encapsuler et un drapeau booléen indiquant l'état de cette donnée. Le fait de stocker une référence permet de limiter les copies intempestives et de garantir le temps d'accès à l'élément encapsulé.
- une série de méthodes de conversion automatique de `data<T>` vers une ré-

¹⁵Ce terme de passerelle est un élément récurrent de génie logiciel plus connu sous le terme anglophone de «*Proxy*» [84]

férence sur `T` afin de rendre transparent et immédiat l'appel d'éventuel méthodes d'instance de `T`. Si l'on tente d'accéder à un élément dont le drapeau de validité est mis à faux, une exception est levée à l'exécution.

- une série de méthodes pour l'envoi et la réception MPI bloquante et la réception non-bloquante. Lors de l'appel de ces méthodes, le drapeau de validité est mis à vrai lorsque les appels aux primitives de réception terminent sans erreur, évitant ainsi l'utilisation de données erronées.

Il reste alors à modifier le comportement des méthodes d'envoi et de réception de la classe `data` pour s'accommoder des divers types de données transmissibles. L'idée est la suivante : les types atomiques et les types composites sont traités de manière homogène par un simple appel à la primitive MPI adéquate. Pour ce faire, une classe *template* annexe permet de lier à un type donné la valeur du `MPI_Datatype` correspondant. Les types composites devront alors exposer une méthode statique `Register` qui permettra, une fois appelée en début de programme, de construire ce `MPI_Datatype`. Pour les types utilisant un protocole spécifique, l'utilisateur devra leur fournir une interface définie par la classe transmissible (listing 5.28).

Listing 5.28 – Interface de la classe transmissible

```

1  class transmissible
2  {
3      public:
4
5      void receive( int src, int tag, MPI_Status& st ) {}
6      void send( int dst, int tag ) const {}
7      void probe( int src, int tag, MPI_Request& req ) {}
8      void ireceive( int src, int tag ) {}
9
10     protected:
11         transmissible() {}
12     };

```

Cette classe est alors utilisée comme une classe abstraite — il est en effet impossible de construire une instance de `transmissible` — mais ne fournit pas de méthode abstraite en tant que telle. Le **type** `transmissible` agit comme un marqueur que la classe `data` va être capable de détecter via un comparateur de type statique (annexe I.3.3). Par exemple, la méthode `send` de la classe `data` s'écrit :

Listing 5.29 – Méthode send de la classe data

```

1  template<class T>
2  void data<T>::send( int dst, int tag ) const
3  {
4      typedef typename is_transmissible<T>::type_t send_t;
5      mpi_comm<T, send_t>::Send(mData, dst, tag);
6  }
7
8  template<class T, class F = false_t>
9  struct mpi_comm
10 {
11     static inline
12     void Send(const T& data, int dst, int tag)
13     {
14         MPI_Send( &data, 1, mpi_info<T>::Datatype,
15                   dst, tag, MPI_COMM_WORLD );
16     }
17 };
18
19 template<class T>
20 struct mpi_comm<T, true_t>
21 {
22     static inline
23     void Send(const T& data, int dst, int tag)
24     {
25         data.send(dst, tag);
26     }
27 };

```

Lorsqu'un appel à `data<T>::send()` est compilé, le compilateur effectue un test pour déterminer si `T` et `transmissible` font partie de la même hiérarchie de tâche. Selon le résultat de ce test, soit le compilateur déroule l'appel correspondant à un envoi atomique ou composite (via `mpi_comm<T, false_t>`), soit il déroule l'appel à la méthode `send` fournie par l'instance du type `T` (via `mpi_comm<T, true_t>`). À aucun moment le système de résolution dynamique des méthodes virtuelles n'entre en jeu, car le test statique sur les types permet à `transmissible` d'agir comme l'équivalent d'une classe abstraite statique.

5.4 Validation

Il convient maintenant de quantifier les performances des applications écrites via l’interface de QUAFF en les comparant avec les temps d’exécution d’un code utilisant seulement les primitives de base fournies par MPI, le but étant de mesurer le surcoût dû au code produit par QUAFF par rapport à la version MPI.

Les protocoles de test utilisés consistent à effectuer des mesures sur 1000 à 10000 exécutions en utilisant des tâches séquentielles «synthétiques» dont la durée d’exécution est connue et paramétrable. Nous avons alors effectué des mesures pour l’ensemble des squelettes parallèles proposés par QUAFF :

- Pour le squelette **Pipeline**, nous avons mesuré les écarts de débit et de latence entre un **Pipeline** équilibré de N étages et son équivalent MPI pour N allant de 2 à 10 et des fonctions dont le temps d’exécution varie entre 0,001s et 0,5s.
- Pour le squelette **Pardo**, nous avons mesuré l’écart entre son temps d’exécution et le temps d’exécution d’un code MPI équivalent. Ce test sera effectué pour des squelettes **Pardo** de N éléments, avec N compris entre 2 et 10 et des temps d’exécutions τ_i compris entre 0,001s et 0,5s.
- Pour le squelette **Farm**, nous nous plaçons dans le cas où les temps de communications sont constants (tous les éléments du flux ont une taille similaire) et où le temps de calcul ne dépend que de la donnée d’entrée. Nous faisons aussi l’hypothèse que le temps d’attente τ_{wait} est négligeable. Nous cherchons alors à évaluer le rapport entre le temps de mise à disposition d’un squelette `farm` fourni par QUAFF et son implantation MPI en faisant varier le temps de calcul des processus esclaves (entre 0,001s et 0,5s) ainsi que le nombre d’esclaves (entre 2 à 10).
- Pour le squelette **SCM**, nous nous plaçons dans un cas où le temps de calcul de chaque processus esclave est fixé et varie entre 0,001s et 0,5s. Nous mesurons alors l’écart entre le ratio calcul/communication de l’implantation QUAFF et du code MPI équivalent pour un nombre de processus esclaves allant de 2 à 10.

Les résultats de ces mesures sont présentés ci-dessous :

- Pour le squelette **Pipeline**, le surcoût sur la latence reste toujours inférieur à 5% quel que soit le nombre d'étages du pipeline. Pour le débit, les mesures du surcoût induit par QUAFF sont indiquées dans le tableau 5.1. On note que le surcoût reste toujours inférieur à 3%, ce qui est très satisfaisant.

τ	N = 2	N = 3	N = 4	N = 5	N = 6	N = 7	N = 8	N = 9	N = 10
1 ms	2.33%	1.75%	1.40%	1.17%	1.00%	0.88%	0.78%	0.70%	0.64%
10 ms	0.58%	0.54%	0.50%	0.47%	0.44%	0.41%	0.39%	0.37%	0.35%
50 ms	0.15%	0.15%	0.14%	0.13%	0.13%	0.12%	0.12%	0.11%	0.12%
100 ms	0.07%	0.07%	0.07%	0.07%	0.06%	0.06%	0.05%	0.04%	0.04%
500 ms	0.01%	0.01%	0.01%	0.01%	0.01%	n/a	n/a	n/a	n/a

TAB. 5.1 – Mesure du surcoût en débit du squelette pipeline

- Pour le squelette **Pardo**, quel que soit la durée des tâches séquentielles et le nombre de processeurs utilisés, le surcoût induit par QUAFF est inférieur à 1% et devient négligeable dès que plus de 5 processeurs sont utilisés. Ce faible surcoût provient du fait qu'aucune communication n'est effectuée entre les divers processus et que le code généré par QUAFF correspond exactement au code MPI équivalent.

τ_i	N = 2	N = 3	N = 4	N \geq 5
1 ms	0.7%	0.1%	2%	< 1%
10 ms	0.1%	3%	< 1%	< 1%
50 ms	4%	< 1%	< 1%	< 1%
> 100 ms	< 1%	< 1%	< 1%	< 1%

TAB. 5.2 – Mesure du surcoût du squelette pardo

- Pour le squelette **Farm**, Le sur-coût induit par QUAFF reste très satisfaisant quel que soit le temps de calcul et reste de l'ordre de 3% quel que soit le nombre de processeurs utilisés.

τ_c	N = 2	N = 3	N = 4	N = 5	N = 6	N = 7	N = 8	N = 9	N = 10
1 ms	1.67%	1.25%	1.00%	0.83%	0.71%	0.63%	0.56%	0.50%	0.45%
10 ms	0.42%	0.38%	0.36%	0.33%	0.31%	0.29%	0.28%	0.26%	0.25%
50 ms	0.11%	0.11%	0.10%	0.10%	0.09%	0.09%	0.09%	0.08%	0.07%
100 ms	0.07%	0.07%	0.07%	0.05%	0.06%	0.05%	0.05%	0.04%	0.02%
500 ms	0.03%	0.02%	0.01%	0.02%	0.01%	n/a	n/a	n/a	n/a

TAB. 5.3 – Mesure du surcoût du squelette farm

- Pour le squelette **SCM**, le surcoût devient plus élevé mais reste dans une fourchette acceptable de 2 à 3%.

τ_f	N = 2	N = 3	N = 4	N = 5	N = 6	N = 7	N = 8	N = 9	N = 10
1 ms	3.35%	2.50%	2.20%	1.87%	1.41%	1.24%	1.10%	1.03%	0.91%
10 ms	0.87%	0.79%	0.71%	0.63%	0.63%	0.59%	0.57%	0.53%	0.50%
50 ms	0.19%	0.21%	0.19%	0.18%	0.17%	0.18%	0.17%	0.15%	0.13%
100 ms	0.11%	0.12%	0.11%	0.10%	0.09%	0.09%	0.08%	0.09%	0.06%
500 ms	0.05%	0.03%	0.04%	0.02%	0.02%	0.01%	n/a	n/a	n/a

TAB. 5.4 – Mesure du surcoût du squelette **scm**

L’ensemble de ces résultats est très satisfaisant. Le surcoût de QUAFF est de l’ordre de 3%. L’origine de ces pertes peut s’expliquer de deux facteurs :

- un surcoût au démarrage des processus de calcul dû à la stratégie d’exécution de chaque squelette;
- un surcoût dû aux communications via les appels aux méthodes de la classe `data`.

Dans le cadre d’une application opérant sur un flux, ce surcoût devient rapidement négligeable lorsque l’on atteint le régime permanent. Dans le cas du squelette **SCM**, le surcoût supplémentaire provient en outre de la légère différence entre le code de diffusion/fusion MPI et celui du squelette. En effet, si le code de **SCM** est écrit de manière générique afin de s’adapter à divers types de données, le code MPI est souvent optimisé pour utiliser des primitives comme `MPI_Scatter`, `MPI_Reduce` et un code optimisé pour un type de donnée particulier.

5.5 Conclusion

La définition et l’implantation d’une bibliothèque de programmation parallèle à base de squelettes ont fait l’objet de nombreux travaux. Les modèles mis en place par ces bibliothèques sont très pertinents et permettent d’obtenir un niveau d’abstraction très élevé et une très grande expressivité. Pourtant, peu de projets ont réussi à concilier ce haut niveau d’abstraction et un niveau de performance comparable à celui obtenu par une implantation directe en MPI.

QUAFF se positionne donc comme une bibliothèque C++ pour la programmation parallèle à base de squelettes qui, comparativement aux travaux similaires, fournit une implantation dont les performances sont équivalentes à celles fournies par MPI sans pour autant sacrifier son expressivité grâce à l'utilisation de techniques de métaprogrammation qui permettent d'effectuer **au moment de la compilation** la majorité des tâches d'optimisation et de déploiement des squelettes.

Une autre manière de juger de la pertinence de QUAFF est de déterminer si elle répond aux exigences énumérées par Cole [44] pour la définition de bibliothèques de squelettes :

- Basée sur un interface simple en C++, QUAFF participe à **la diffusion du concept de squelette avec un minimum de changements**. Ceci est particulièrement vrai dans le domaine de la vision où de nombreux développeurs sont souvent peu enclins à changer leur habitudes de développement et à devoir réécrire des codes déjà disponibles.
- QUAFF propose un **support pour le parallélisme *ad hoc***, soit en utilisant le squelette **Pardo**, soit en insérant des appels MPI directement au sein des fonctions définies par l'utilisateur.

Il reste alors à démontrer que QUAFF permet **d'exhiber un gain notable dans le temps de développement d'applications parallèles de complexité plus réaliste**. Pour ce faire, nous nous proposons dans le chapitre suivant d'étudier la mise au point de telles applications.

Chapitre 6

Applications à la stéréovision

«Around computers it is difficult to find the correct unit of time to measure progress. Some cathedrals took a century to complete. Can you imagine the grandeur and scope of a program that would take as long?»

Epigrams in Programming #28, ACM SIGPLAN 1982

Grâce aux travaux présentés aux chapitres 3 et 4, nous disposons désormais, d'une part, d'une machine parallèle hybride permettant d'atteindre des accélérations importantes avec un nombre relativement faible de nœuds de calcul et, d'autre part, d'un ensemble d'outils de développement de programmes parallèles autorisant l'expression d'algorithmes de grande complexité tout en conservant des performances élevées. Ce chapitre illustre l'interaction entre ces deux éléments constitutifs de la plate-forme BABYLON et montre comment des applications de vision artificielle complexes peuvent être implantées sur cette architecture.

Le spectre des applications de vision artificielle que nous pourrions aborder est extrêmement vaste. Nous nous sommes délibérément focalisés sur des problématiques dont l'implantation séquentielle temps réel nécessite un grand nombre de compromis en terme de précision, de quantité de données ou de robustesse et dont la validation qualitative a déjà fait l'objet de travaux au sein de notre groupe de travail. D'autre part, comme le capteur équipant la plate-forme BABYLON est composé de deux caméras calibrées, nous nous sommes orientés vers des applications de vision stéréoscopique .

Nous allons donc dans un premier temps aborder les notions nécessaires à la compréhension des problématiques de la stéréo-vision puis nous présenterons

successivement deux applications — une application de reconstruction 3D et une application de suivi de personne — en nous attardant sur les algorithmes sous-jacents, leur implantation séquentielle, leur parallélisation et leur performance temporelle.

6.1 Principe de la stéréo-vision

Une des problématiques abordées par la vision artificielle est de reconstruire des représentations tridimensionnelles de scènes ou d'objets dont on ne possède *a priori* que des représentations projectives en deux dimensions sous forme d'images de diverses origines. Pour cela, plusieurs techniques ont été développées :

- le «*shape from shading*» [1, 113] ou photoclinométrie qui consiste à reconstruire le relief d'une scène à partir d'une seule image de cette scène, en utilisant, en chaque point de l'image, la relation entre le niveau de gris et l'orientation locale de la surface et en résolvant l'équation dite eikonale [62, 114], qui est une équation aux dérivées partielles du premier ordre, non linéaire;
- les approches dites multi-vues mettent en oeuvre plusieurs images (souvent un très grand nombre) représentant une même scène sous divers angles de vues [148, 139]. On peut alors extraire une représentation tridimensionnelle de la scène sous forme d'un nuage quasi-dense [122], d'une carte de disparité [25] ou d'une enveloppe convexe [81];
- les approches binoculaires — c'est à dire utilisant seulement deux capteurs images — qui utilisent les informations comme les positions et les paramètres internes des capteurs afin de résoudre géométriquement les équations de triangulation des zones d'intérêts extraites d'une paire d'image;

Parmi ces diverses variantes, nous nous focaliserons sur la **stéréo-vision binoculaire**, c'est à dire les algorithmes permettant d'extraire des informations de volume à partir d'une paire d'images en provenance de caméras numériques calibrées. La stéréo-vision binoculaire fournit en effet un ensemble d'outils efficaces [95] qui, depuis quelques années, se sont révélés pertinents [96]. Nous allons donc présenter, dans un premier temps, les outils géométriques nécessaires à l'implantation de nos applications.

6.1.1 Modèle sténopé des caméras

Il existe plusieurs modélisations du fonctionnement des caméras. Nous ne considérerons que le modèle *pin-hole* ou **sténopé** qui reste le plus utilisé. Dans ce modèle, une caméra est un outil projectif défini par la position de son centre optique \mathbf{C} et un plan image \mathbf{P}_i . Pour tout point $\mathbf{P} \neq \mathbf{C}$ appartenant au repère du monde \mathbf{R}_w , on associe un point \mathbf{I}_p appartenant au plan image de la caméra (figure 6.1). Cette relation se modélise par une application projective et est représentée par une matrice P_{34} 3×4 de rang 3 connue à un facteur près.

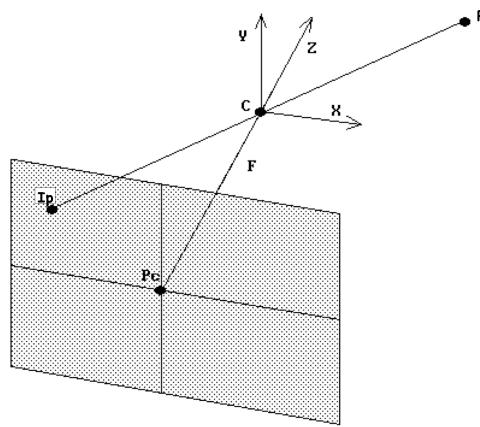


FIG. 6.1 – Modèle sténopé monoculaire

La projection \mathbf{I}_p d'un point \mathbf{P} du repère monde dans le repère image est alors donnée par les relations¹ :

$$\mathbf{I}_p = \begin{pmatrix} u_p \\ v_p \\ 1 \end{pmatrix} \quad \mathbf{P} = \begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix}$$

$$\mathbf{I}_p \propto P_{34}\mathbf{P}$$

Dans le cas général, on peut décrire géométriquement une caméra en spécifiant des paramètres purement euclidiens et des paramètres purement projectifs. On parle alors de modèle calibré et on peut décomposer la matrice P_{34} en un produit KM dans lequel M est la matrice des paramètres extrinsèques :

$$M = \begin{pmatrix} R & T \\ 0_{1 \times 3} & 1 \end{pmatrix}$$

¹Tout au long de ce chapitre, nous utiliserons les notations des points et vecteurs en coordonnées homogènes.

dans laquelle R est une matrice de rotation et T un vecteur de translation qui donnent la pose de la caméra dans le repère du monde et où K est la matrice des paramètres intrinsèques :

$$K \approx \begin{pmatrix} \alpha f & \tau & u_0 & 0 \\ 0 & f & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

dans laquelle :

- f représente la distance focale de la caméra, c'est à dire la distance entre le centre optique et le plan image;
- α le rapport d'aspect, c'est à dire le coefficient de proportionnalité entre la largeur et la hauteur des pixels du capteur;
- τ le défaut d'orthogonalité au plan image de l'axe de vue;
- (u_0, v_0) les coordonnées dans le plan image de son intersection avec l'axe optique.

6.1.2 Principes géométrique de la stéréo-vision

À partir de ce modèle, la construction d'une paire stéréoscopique consiste à coupler deux caméras et à expliciter les relations géométriques entre ces dernières (fig. 6.2).

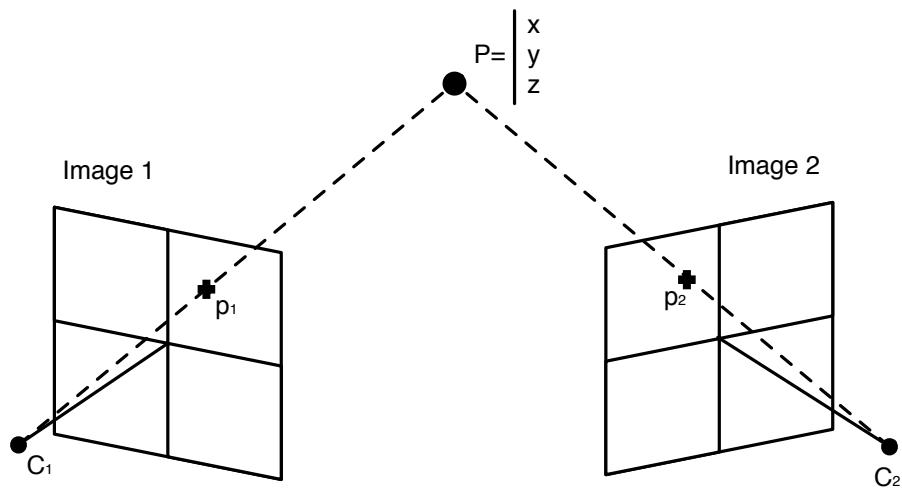


FIG. 6.2 – Géométrie d'une paire stéréoscopique

L'estimation des coordonnées (x, y, z) d'un point P dans le repère du monde est alors possible dès que l'on possède la paire de points — un dans chaque image — provenant de la projection de P dans chaque repère image. Pour ce faire, on exprime analytiquement le fait que chaque point 3D se projette en un point dans le repère image de chaque caméra. Dans le cas général, cela revient à résoudre un système sur-dimensionné de quatre équations à trois inconnues :

$$\begin{cases} K_1 M_1 \mathbf{P} = s_1 \mathbf{I}_1 \\ K_2 M_2 \mathbf{P} = s_2 \mathbf{I}_2 \end{cases}$$

où K_i et M_i représentent les matrices des paramètres intrinsèques et extrinsèques de chaque caméra, \mathbf{P} le point 3D à trianguler, $\mathbf{I}_1, \mathbf{I}_2$ les projections de \mathbf{P} dans chacun des repères images et s_1, s_2 les facteurs d'échelles appliqués aux coordonnées homogènes de des derniers.

6.2 Reconstruction 3D temps réel

La première des applications implantées sur BABYLON a pour but de construire un nuage de points 3D provenant de la reconstruction d'une scène filmée par une paire stéréoscopique [73]. Cette application de reconstruction est extrêmement classique et met en œuvre une chaîne de traitements constituée de trois étapes :

- une étape d'extraction de primitives qui va détecter au sein des paires d'images en provenance des deux caméras des primitives — points, lignes, zones de couleurs — que l'on va chercher à apparier;
- une étape de mise en correspondance qui vise à apparier les primitives extraites de l'image de gauche avec les primitives de l'image de droite (et réciproquement) en utilisant une fonction de mesure qui permet de décider de manière sûre si deux primitives représentent bien la projection d'un même objet de la scène 3D;
- une étape de triangulation qui, grâce aux coordonnées de ces primitives, calcule leur position dans le repère du monde.

Notre implantation ajoute à cette chaîne deux étapes de pré-traitement. La première de ces opérations consiste à corriger la distorsion radiale due aux optiques des caméras. La deuxième opération, la rectification épipolaire, consiste à simplifier la géométrie du problème afin de simplifier l'étape d'appariement.

6.2.1 Implantation séquentielle

Pour effectuer cette reconstruction, nous allons appliquer cette chaîne de traitement à chaque paire d'images provenant de la base stéréoscopique. Nous effectuons une étape d'extraction de primitives en utilisant le détecteur de points d'intérêts de Harris et Stephen [94], une étape de mise en correspondance par corrélation croisée et une étape de triangulation. Comme indiqué plus haut, ces étapes sont précédées par une phase de correction de la distorsion radiale et de rectification epipolaire.

6.2.1.1 Correction de la distorsion radiale

Les distorsions radiales sur une image proviennent en général de plusieurs sources : les défauts de courbures des lentilles des caméras, un mauvais alignement de ces lentilles au sein du capteur et l'inclinaison entre ces dernières. Plusieurs modèles de distorsions ont été proposé. Nous utiliserons ici un modèle simple qui consiste à évaluer la position réelle d'un point de l'image en fonction de sa distance au centre de l'image (figure 6.3).

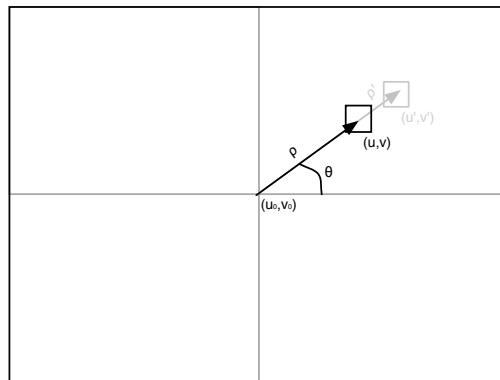


FIG. 6.3 – Correction de la distorsion radiale

Dans ce modèle, la position réelle d'un pixel, donnée par le couple (ρ, θ) , s'évalue en fonction de la position mesurée (ρ', θ') . En pratique, on a :

$$\rho' \begin{bmatrix} \cos \theta' \\ \sin \theta' \end{bmatrix} = \rho (1 + \lambda_1 \rho^2 + \lambda_2 \rho^4) \begin{bmatrix} \cos \theta \\ \sin \theta \end{bmatrix}$$

où λ_1 et λ_2 sont les paramètres caractérisant la distorsion obtenus lors de la calibration et :

$$\rho = \sqrt{(u - u_0)^2 + (v - v_0)^2}$$

Comme cette correction ne dépend que des paramètres issues de la calibration des caméras [121], une table contenant les corrections à appliquer à chaque pixel est pré-calculée.

6.2.1.2 Rectification épipolaire

Sur une paire d'images stéréoscopiques, la rectification épipolaire consiste à déterminer une transformation de chaque plan image telle que les paires de droites épipolaires conjuguées soient situées sur la même droite et parallèles à un des axes du repère image. Les images rectifiées peuvent alors être considérées comme provenant d'une nouvelle paire stéréoscopique virtuelle, obtenue en effectuant une rotation des caméras initiales. L'avantage de la rectification est de simplifier la mise en correspondance de points entre les deux images en la limitant à des paires de lignes horizontales au sein des images rectifiées. Plusieurs méthodes de rectification ont été proposées [21, 144, 95] mais peu utilisent de façon efficace les contraintes de calibration d'une paire stéréoscopique. Fusiello propose un algorithme simple [83] permettant d'effectuer la rectification d'une paire d'images stéréoscopiques provenant d'une base stéréo calibrée.

Soit une paire de caméras calibrées dont nous connaissons les matrices des paramètres intrinsèques (K_1 et K_2) et les matrices des paramètres extrinsèques (M_1 et M_2). L'algorithme proposé par Fusiello consiste à définir deux nouvelles caméras virtuelles dont les paramètres extrinsèques sont obtenus par l'application d'une rotation sur les caméras composant la paire stéréoscopique initiale. Cette rotation permet alors de rendre les plans images coplanaires. En outre, ces nouveaux plans images devront être parallèles à la droite joignant les centres optiques des deux caméras (figure 6.4).

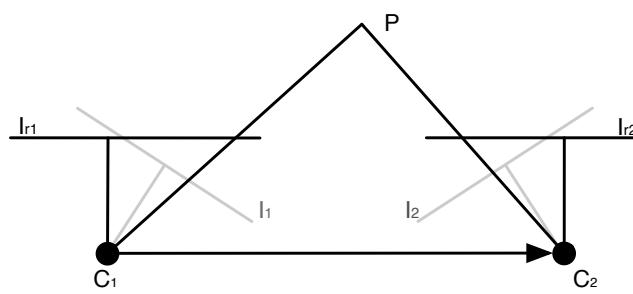


FIG. 6.4 – Principe de la rectification épipolaire

Enfin, afin d'assurer que la rectification est correcte, les deux nouvelles caméras utilisent les mêmes paramètres intrinsèques. L'algorithme proposé par Fusiello permet ainsi d'obtenir à la fois les paramètres extrinsèques et intrinsèques des caméras rectifiées ainsi que l'homographie permettant de passer du plan image original au plan image rectifié.

Soit R_2 et R_1 , les matrices de rotation donnant la pose des caméras droite et gauche dans le repère du monde; T_2 et T_1 , la position de ces caméras dans ce même repère et K_2 et K_1 , les matrices des paramètres intrinsèques des caméras. L'algorithme évalue une nouvelle rotation qui va définir une nouvelle paire de plans images parallèles. Pour ce faire, il est nécessaire de construire une base orthonormée à partir des paramètres extrinsèques de la base stéréoscopique initiale. Nous définissons donc un vecteur \mathbf{X} unitaire qui s'appuie sur la ligne épipolaire originale.

$$\mathbf{X} = \frac{\mathbf{R}_2' \mathbf{T}_2 - \mathbf{R}_1' \mathbf{T}_1}{\|\mathbf{R}_2' \mathbf{T}_2 - \mathbf{R}_1' \mathbf{T}_1\|}$$

Le vecteur \mathbf{Y} est obtenu via le produit vectoriel d'un vecteur \mathbf{k} , choisi arbitrairement de manière à ce que \mathbf{Y} appartienne à un plan orthogonal à \mathbf{k}^2 et \mathbf{X} :

$$\mathbf{Y} = \frac{\mathbf{k} \wedge \mathbf{X}}{\|\mathbf{k} \wedge \mathbf{X}\|}$$

Finalement, l'axe \mathbf{Z} est obtenu par le produit vectoriel de \mathbf{X} et \mathbf{Y} :

$$\mathbf{Z} = \frac{\mathbf{X} \wedge \mathbf{Y}}{\|\mathbf{X} \wedge \mathbf{Y}\|}$$

Ces trois vecteurs forment alors une base orthonormale, parallèle à la ligne épipolaire initiale. On construit alors la matrice de rotation R_r :

$$R_r = (\mathbf{X} \quad \mathbf{Y} \quad \mathbf{Z})$$

et nous l'utilisons pour calculer la pose des caméras virtuelles :

$$\mathbf{M}_{r_1} = \begin{pmatrix} R_r & -R_2' \mathbf{T}_2 \\ 0_{1 \times 3} & 1 \end{pmatrix}$$

$$\mathbf{M}_{r_2} = \begin{pmatrix} R_r & -R_1' \mathbf{T}_1 \\ 0_{1 \times 3} & 1 \end{pmatrix}$$

²En pratique, nous utilisons l'axe $\mathbf{C}_1 \mathbf{Z}$ d'une des caméras.

Il reste alors à définir les paramètres intrinsèques de ces caméras. Pour ce faire, nous prenons simplement la moyenne des paramètres intrinsèques des caméras initiales et nous fixons le défaut d'orthogonalité à 0.

$$K_r = \frac{1}{2}(K_2 + K_1)$$

Les nouvelles matrices de projection des caméras rectifiées sont alors données par la relation :

$$\begin{aligned} P_{r_1} &= K_r M_{r_1}^{-1} \\ P_{r_2} &= K_r M_{r_2}^{-1} \end{aligned}$$

Il ne nous reste plus qu'à évaluer la transformation amenant un pixel d'une image issue d'une des caméras vers le pixel correspondant dans l'image rectifiée. Pour ce faire, nous exprimons P_1 et P_2 , les matrices de projections des caméras initiales, sous la forme :

$$P_1 = (Q_1 \quad q_1) \quad P_2 = (Q_2 \quad q_2)$$

et nous exprimons P_{r_1} et P_{r_2} comme :

$$P_{r_1} = (Q_{r_1} \quad q_1) \quad P_{r_2} = (Q_{r_2} \quad q_2)$$

Les homographies permettant de passer d'un pixel de l'image originale à l'image rectifiée est alors donnée par :

$$H_1 = Q_1 Q_{r_1}^{-1} \quad H_2 = Q_2 Q_{r_2}^{-1}$$

Ainsi, si l'on considère un point p_{r_1} , projection de P dans une image rectifiée, son homologue p_{o_1} dans l'image originale se calcule par la relation :

$$\mathbf{p}_{o_1} \propto H_1 \mathbf{p}_{r_1}$$

Dans le cas général, les coordonnées de \mathbf{p}_{o_1} ne sont pas entières. Une étape d'interpolation bilinéaire est nécessaire pour obtenir l'intensité effective de \mathbf{p}_{r_1} . La figure 6.5 illustre l'application de cette méthode sur une paire d'images issue de notre base stéréoscopique, avec à gauche, les deux images originales et à droite les images rectifiées. Un faisceau de droites est tracé en surimpression afin de mettre en évidence l'alignement des points homologues.

Comme pour la correction de la distorsion radiale, la transformation de rectification est pré-calculée sous forme d'une table de mise en correspondance. Elle est d'ailleurs fusionnée avec la table de correction de la distorsion afin d'effectuer ces deux corrections en une seule passe.

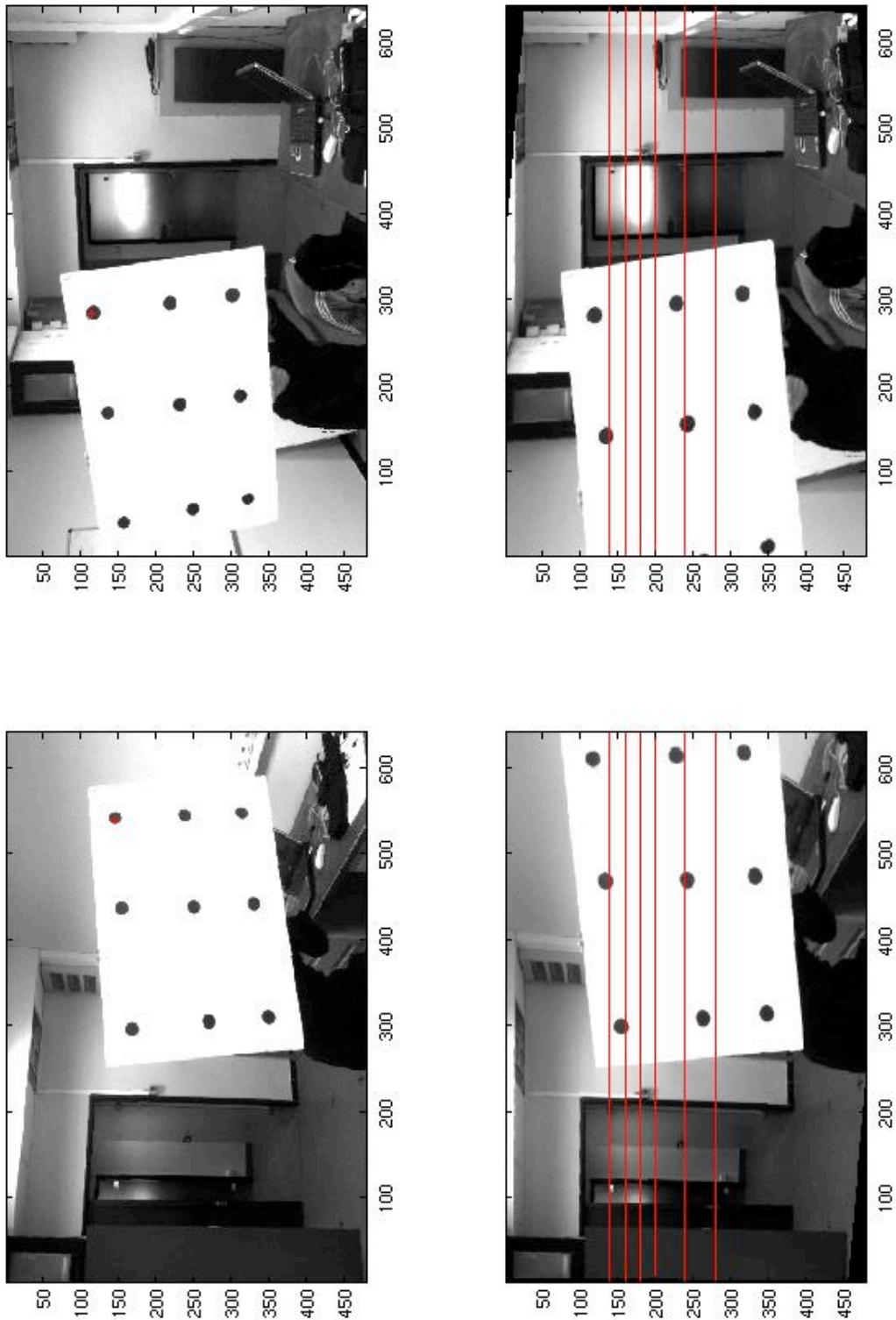


FIG. 6.5 – Algorithme de rectification - Plans image rectifiés [83]

6.2.1.3 Détection des points d'intérêt

La détection de points d'intérêt est, au même titre que la détection de contours, une étape préliminaire à de nombreux processus de vision par ordinateur. Les points d'intérêt, dans une image, correspondent à des discontinuités dans les deux directions de l'intensité des pixels. Ce sont par exemple les coins ou les points de fortes variations de texture. De nombreuses méthodes de détection ont été proposées et parmi celles ci, on trouve le détecteur de Moravec [134] et le détecteur de Harris et Stephen [94].

Détecteur de Moravec

L'idée du détecteur de Moravec est de considérer le voisinage d'un pixel (une fenêtre) et de déterminer les changements moyens de l'intensité dans le voisinage considéré lorsque la fenêtre se déplace dans diverses directions. Plus précisément, on considère la fonction :

$$E(x, y) = \sum_{u,v} w(u, v) |I(x+u, y+v) - I(x, y)|^2$$

où w spécifie le voisinage considéré (valeur 1 à l'intérieur de la fenêtre et 0 à l'extérieur), $I(u, v)$ est l'intensité du pixel de coordonnées (u, v) et $E(x, y)$ représente la variance du changement d'intensité lorsque la fenêtre est déplacée de (x, y) .

Trois situations peuvent alors survenir :

- L'intensité est approximativement constante dans la zone image considérée et la fonction E prendra alors de faibles valeurs dans toutes les directions (x, y) ;
- La zone image considérée contient un contour rectiligne. E prendra alors de faibles valeurs pour des déplacements (x, y) le long du contour et de fortes valeurs pour des déplacements perpendiculaires au contour.
- La zone image considérée contient un coin ou un point isolé : E prendra de fortes valeurs dans toutes les directions.

Le principe du détecteur de Moravec est donc de rechercher les maxima locaux de la valeur minimale de E en chaque pixel et dont la valeur est supérieure à un seuil fixé arbitrairement.

Détecteur de Harris et Stephen

Le détecteur de Moravec souffre néanmoins de nombreuses limitations. Harris et Stephen ont identifié certaines de ces limitations et, en les corrigeant, en ont déduit un détecteur de coins plus efficace. Tout d'abord, la réponse du détecteur est anisotropique en raison du caractère discret des directions de changement que l'on peut effectuer (des pas de 45 degrés). Pour améliorer cet aspect, il suffit de considérer le développement de Taylor de la fonction d'intensité I au voisinage du pixel (u, v) :

$$I(x+u, y+v) = I(u, v) + x \frac{\partial I}{\partial x} + y \frac{\partial I}{\partial y} + o(\sqrt{x^2 + y^2})$$

D'où

$$E(x, y) = \sum_{u, v} w(u, v) \left| x \frac{\partial I}{\partial x} + y \frac{\partial I}{\partial y} + o(\sqrt{x^2 + y^2}) \right|^2$$

Pour des déplacements de faible intensité, on néglige $o(\sqrt{x^2 + y^2})$ pour obtenir l'expression suivante :

$$E(x, y) = Ax^2 + 2Cxy + By^2$$

Avec³ : $A = \left(\frac{\partial I}{\partial x} \right)^2 * w$, $B = \left(\frac{\partial I}{\partial y} \right)^2 * w$ et $C = \left(\frac{\partial I}{\partial x} \frac{\partial I}{\partial y} \right) * w$.

Ensuite, la réponse du détecteur de Moravec est bruitée en raison du voisinage considéré. Le filtre w utilisé est en effet binaire (valeur 0 ou 1) et est appliqué sur un voisinage rectangulaire. Pour éviter cela, Harris et Stephen ont proposé d'utiliser un filtre Gaussien :

$$w(u, v) = e^{-\frac{u^2+v^2}{2\sigma^2}}$$

Enfin, le détecteur de Moravec répond de manière trop forte aux contours en raison du fait que seul le minimum de E est pris en compte en chaque pixel. Pour prendre en compte le comportement général de la fonction E localement, on écrit :

$$E(x, y) = (x, y) M(x, y)^t$$

Avec :

$$M = \begin{bmatrix} A & C \\ C & B \end{bmatrix}$$

La matrice M caractérise le comportement local de la fonction E et les valeurs propres de cette matrice correspondent aux courbures principales associées à E :

³* représente le produit tensoriel

- Si les deux courbures sont de faible valeur, alors la région considérée a une intensité approximativement constante.
- Si une des courbures est de forte valeur alors que l'autre est de faible valeur alors la région contient un contour.
- Si les deux courbures sont de forte valeur alors l'intensité varie fortement dans toutes les directions, ce qui caractérise un coin.

Harris et Stephen ont alors proposé l'opérateur suivant pour détecter les coins dans une image :

$$R = \text{Det}(M) - k \text{Trace}(M)^2$$

où

$$\text{Det}(M) = AB - C^2, \quad \text{et} \quad \text{Trace}(M) = A + B$$

Les valeurs de R sont alors positives au voisinage d'un coin, négatives au voisinage d'un contour et faibles dans une région d'intensité constante. Après un seuillage sur les valeurs de R, une recherche des maxima locaux de R permet donc d'extraire les points présentant le plus d'intérêt.

La figure 6.6 présente le résultat de cet algorithme sur une image issue de notre système d'acquisition. Elle représente les points d'intérêt détectés en surimpression de l'image originale.



FIG. 6.6 – Résultat de l'application d'un détecteur de Harris et Stephen

6.2.1.4 Mise en correspondance des points d'intérêt

Reste à effectuer les appariements : comment rechercher dans une image le point homologue d'un point donné dans l'autre ? Cette recherche est facilitée par des contraintes impliquées par l'hypothèse de rigidité du couple des deux caméras. Un point P projeté en p_1 dans la caméra 1 appartient à la droite $C_1 p_1$. Cette droite se projette dans l'image de la caméra 2 en une droite appelée **droite épipolaire** associée à u . En conséquence l'homologue de p_1 — le point p_2 — se trouvera sur cette droite. Cette zone de recherche se restreint alors à une ligne de pixels au sein de chaque image. En outre, en travaillant directement sur les images rectifiées, pour chaque ligne de pixel de l'image 1, sa ligne homologue dans l'image 2 est tout simplement la ligne de même ordonnée (figure 6.7).

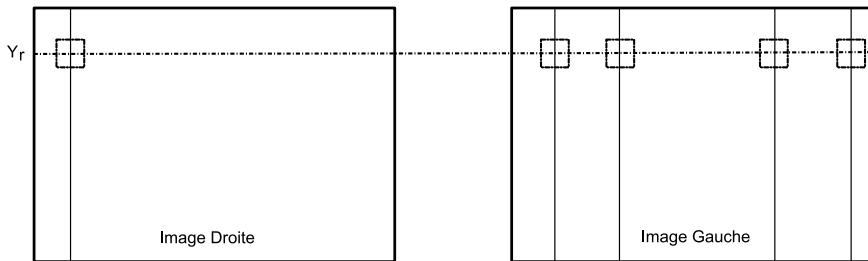


FIG. 6.7 – Stratégie d'appariement sur une droite épipolaire

Une fois que l'espace de recherche est restreint, on détermine si un point p_2 est homologue à chaque point p_1 en utilisant un score de corrélation. En pratique, ce score doit permettre d'identifier clairement des appariements dans des conditions d'éclairage ou d'orientation de caméras différents. Dans notre cas, en travaillant sur les images rectifiées, les seules sources d'erreurs proviennent des variations de luminosité entre les images. Le critère retenu est alors une corrélation centrée normée — ou ZNCC⁴. On définit la ZNCC entre deux voisinages des images A et B, centrés en (x, y) et (x', y') et de taille $K \times K$ comme le rapport entre la covariance de ces deux voisinages et le produit de leur écart-type :

$$\text{ZNCC}_K(A(x, y), B(x', y')) = \frac{\text{cov}(A(x, y), B(x', y'))}{\sigma(A(x, y))\sigma(B(x', y'))}$$

L'algorithme d'appariement consiste alors à calculer le score de corrélation entre chaque point d'intérêt de l'image A et les points d'intérêt de l'image B possédant

⁴Zero Normalized Cross-Correlation

la même ordonnée. En calculant ces scores, nous ne conservons que les scores supérieurs à un seuil (typiquement 0.9) et nous sélectionnons le meilleur d'entre eux ainsi que son point associé. Au final, le couple de point (p_A, p_B) ayant reçu le meilleur score de corrélation est conservé comme paire de points appariés.

6.2.1.5 Triangulation

Une fois les points d'intérêt appariés, il ne nous reste plus qu'à effectuer leur triangulation. Soit un point P :

$$\mathbf{P} = \begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix}$$

Ses projections dans les caméras droite et gauche sont respectivement les points I_1 et I_2 .

$$\mathbf{I}_1 = s_1 \begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix} \quad \mathbf{I}_2 = s_2 \begin{pmatrix} u_2 \\ v_2 \\ 1 \end{pmatrix}$$

Comme nous l'avons vu au paragraphe 6.1.2, cette relation revient à résoudre le système :

$$\begin{cases} \mathbf{I}_1 = \mathbf{C}_1 \mathbf{P} \\ \mathbf{I}_2 = \mathbf{C}_2 \mathbf{P} \end{cases}$$

Avec :

$$\mathbf{C}_1 = \begin{pmatrix} \mathbf{C}_{11}^T \\ \mathbf{C}_{12}^T \\ \mathbf{C}_{13}^T \end{pmatrix} \quad \mathbf{C}_2 = \begin{pmatrix} \mathbf{C}_{21}^T \\ \mathbf{C}_{22}^T \\ \mathbf{C}_{23}^T \end{pmatrix}$$

où \mathbf{C}_{ij}^T représente la j ^{ème} ligne de la matrice \mathbf{C}_i . En éliminant les facteurs d'échelles s_1 et s_2 de ce système, on se ramène à un système sur-dimensionné — 4 équations pour 3 inconnues — qui se résout classiquement par un calcul de noyau.

$$\begin{cases} (\mathbf{C}_{11}^T - u_1 \mathbf{C}_{13}^T) \cdot \mathbf{P} = 0 \\ (\mathbf{C}_{12}^T - v_1 \mathbf{C}_{13}^T) \cdot \mathbf{P} = 0 \\ (\mathbf{C}_{21}^T - u_2 \mathbf{C}_{23}^T) \cdot \mathbf{P} = 0 \\ (\mathbf{C}_{22}^T - v_2 \mathbf{C}_{23}^T) \cdot \mathbf{P} = 0 \end{cases}$$

Géométriquement, cela revient à déterminer les coordonnées du milieu de la perpendiculaire reliant les deux rayons optiques reliant les centres optiques de chaque caméra au point \mathbf{P} .

6.2.1.6 Performances

Le tableau 6.1 présente le temps d'exécution en millisecondes de cet algorithme de reconstruction exécuté sur un seul processeur d'un nœud de BABYLON⁵. La donnée d'entrée est une paire d'images 640×480 en niveau de gris issues de la base stéréoscopique équipant BABYLON. Nous fixons aussi le nombre de points en sortie du détecteur de Harris à 1000. Le résultat de cette reconstruction est alors donné sur la figure 6.8.

N	Rectif.	Harris	Appar.	Triang.	Total
500	243.4ms	192.3ms	107.3ms	91.3ms	634.3ms
1000	246ms	262ms	304.2ms	180ms	992.2ms
2000	244.3ms	304.1ms	858.6ms	362.1ms	1771.1ms

TAB. 6.1 – Temps d'exécution de la reconstruction 3D pour 1 paire d'image sur un seul processeur

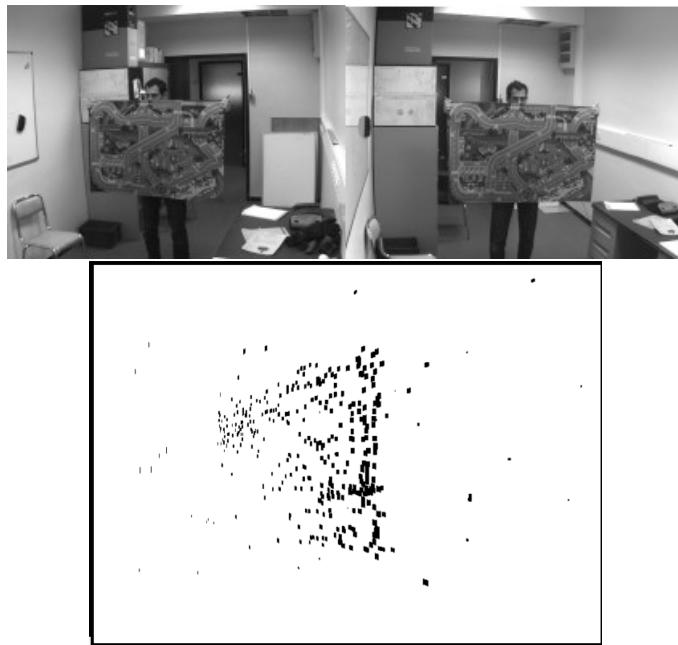


FIG. 6.8 – Résultats d'une reconstruction 3D

⁵Soit une machine POWER PC G5 cadencé à 2GHz équipée de 1Go de mémoire.

6.2.2 Implantation parallèle

Les temps d'exécution mesurés au paragraphe précédent montrent que la reconstruction 3D à la volée ne peut se faire à une vitesse compatible avec la fréquence d'acquisition de ces derniers — c'est à dire à la vitesse de 30ms par paire d'images. Nous allons proposer ici une implantation parallèle de cet algorithme et évaluer ses performances.

6.2.2.1 Analyse du problème séquentiel

Comme pour l'application de stabilisation présentée au chapitre 2, nous nous intéressons à l'importance relative de chaque étape de la chaîne de traitement dans le temps d'exécution total. Le tableau 6.2 récapitule l'analyse des temps d'exécution de chacune de ces étapes.

Etapes	500 pts	1000 pts	2000 pts
Rectification	38.37%	24.79%	13.79%
Détection des points d'intérêt	30.31%	26.40%	17.17%
Mise en correspondance	16.91%	30.65%	48.47%
Triangulation	14.41%	18.16%	20.57%

TAB. 6.2 – Importance relative des étapes de la reconstruction 3D

On remarque que la répartition des temps d'exécution des différentes étapes est relativement équilibrée pour un faible nombre de points. Lorsque le nombre de points à apparié augmente, on note que l'étape de mise en correspondance devient prépondérante. Cette répartition va donc nous conduire à effectuer la parallelisation des quatre étapes constituant cette application.

En terme de type de parallélisme, deux points importants se dégagent :

- Les étapes de rectification et de détection de points d'intérêt peuvent s'exécuter de manière indépendante sur les images droite et gauche du flux vidéo.
- Grâce à l'étape de rectification, les points détectés au sein d'une bande horizontale quelconque sont garantis d'être appariés avec un point issu de la bande correspondante dans l'image homologue. De même, la triangulation d'un ensemble de points issus d'une bande donnée se fait indépendamment de la triangulation des points issus d'une autre bande.

6.2.2.2 Proposition d'implantation

A partir de ces considérations, l'implantation retenue est la suivante :

- Chaque nœud extrait une bande horizontale des images en provenance du réseau FireWire et y applique les opérations de rectification. Les bandes issues des images droite et gauche sont traitées respectivement par un des deux processeurs du nœud via pThread afin de profiter du parallélisme SMP. Compte tenu de la forme de l'algorithme, aucune opération SIMD n'est utilisable pour cette étape.
- Nous détectons ensuite dans chaque paire de bandes des points d'intérêt via le détecteur de Harris et Stephen. De la même manière, chaque processeur détecte ses points d'intérêt séparément via pThread. Ici, E.V.E. permet de vectoriser les étapes de filtrage et de seuillage du détecteur de Harris.
- Pour la mise en correspondance, nous utilisons le fait que les lignes épipolaires de chaque image sont alignées afin de restreindre l'amplitude de la recherche. Comme nous sommes assurés que tous les points d'une bande trouveront leur correspondant éventuel dans la bande équivalente sur l'autre image, il n'est pas nécessaire de fusionner les listes de points d'intérêt en amont de la mise en correspondance, ce qui réduit donc les communications. Ici, E.V.E. est utilisé pour vectoriser les calculs de ZNNC, pendant que les deux processeurs se partagent les calculs à effectuer sur la liste des points appariés.
- Enfin, la reconstruction 3D des points retenus dans chaque bande utilise une implantation SMP-SIMD grâce à l'interface entre E.V.E. et LAPACK.

Le tableau 6.3 résume les différentes étapes de cet algorithme et le type de parallélisme utilisé pour chacune d'entre elle.

Étapes	SIMD	SMP	MIMD
Rectification	-	✓	✓
Détection	✓	✓	✓
Mise en correspondance	✓	✓	✓
Triangulation	✓	✓	✓

TAB. 6.3 – Parallélisation de l'algorithme de reconstruction 3D

Ce schéma de parallélisation est résumé dans la figure 6.9.

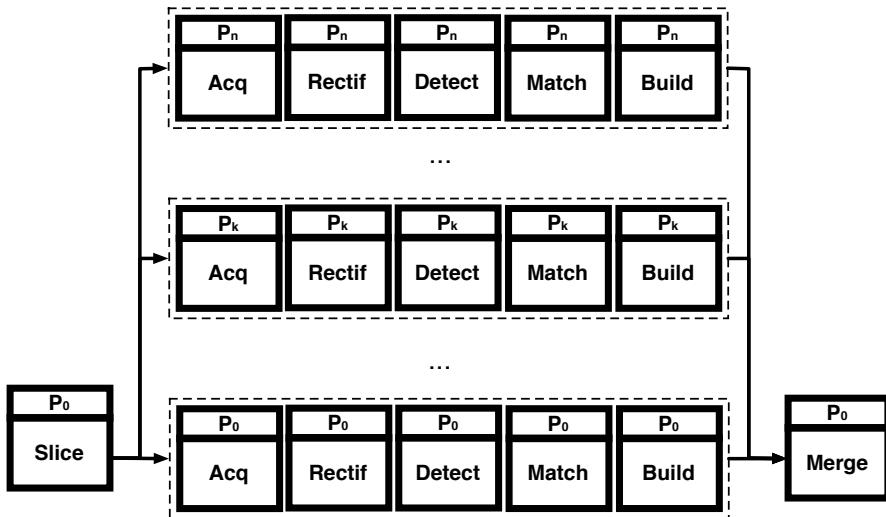


FIG. 6.9 – Graphe de processus communiquant pour l’application de reconstruction 3D

On reconnaît clairement ici l’imbrication d’un squelette **Sequence** contenant les fonctions effectives de l’algorithme au sein d’un squelette **SCM**. Son expression sous forme de squelette se représente alors comme sur la figure 6.10.

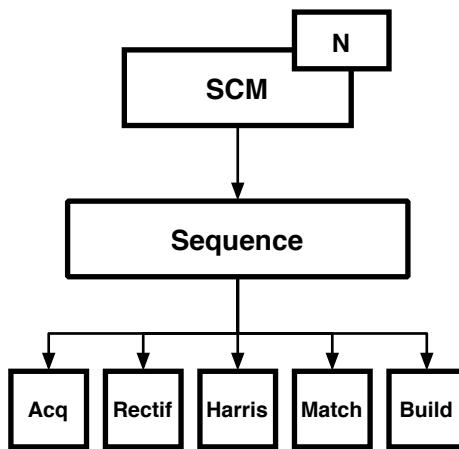


FIG. 6.10 – Schéma d’imbrication des squelettes de l’application de reconstruction 3D

Le point crucial de cette implantation réside dans l'étape de diffusion qui, au lieu de transmettre des bandes d'images sur le réseau, ne transmet que l'indice de la bande (de 0 à N) aux processus esclaves, les données images étant déjà disponible sur les nœuds grâce à la diffusion sur le bus Firewire.

Le code QUAFF de cette application se décompose en trois parties :

- l'intégration des fonctions utilisateur via la construction task.

```

1  typedef task<GetSlices, int, Image>           acq;
2  typedef task<Rectif, Image, Data>            rectif;
3  typedef task<Harris, Data, Data>          detect;
4  typedef task<Match, Data, list<paire>>    match;
5  typedef task<Build, list<paire>, list<p3D>> build;
```

Dans ces définitions, les types Image, p3D, paire et Data sont des types définis par l'utilisateur qui représentent respectivement un conteneur pour les images en provenance des caméras, un point 3D, une paire de coordonnées au sein des repères images et une structure permettant de transférer d'un bloc les bandes d'images nécessaires à la rectification et à la mise en correspondance.

- l'intégration des fonctions de diffusion et récupération du squelette **SCM**.

```

1  typedef task<Slice, none_t, vector<int>>      slice;
2  typedef task<Merge, vector<list<p3D>>, none_t> merge;
```

On remarque ici l'utilisation de la classe standard `vector` comme conteneur pour les éléments à diffuser.

- la définition du squelette **Séquence**.

```

1  typedef stage<acq, rectif, detect, match, build> steps;
2  typedef sequence<steps>                      algo;
```

- la définition du squelette **SCM** et de l'application finale.

```

1  typedef scm<slice, worker<algo, N>, merge> app;
2  typedef application<app>                   recon3d;
```

Le code complet de l'application est donnée dans le listing 6.1.

Listing 6.1 – Implantation QUAFF de l’application de reconstruction 3D

```

1 typedef task<GetSlices, int, Image> acq;
2 typedef task<Rectif, Image, Data> rectif;
3 typedef task<Harris, Data, Data> detect;
4 typedef task<Match, Data, list<paire>> match;
5 typedef task<Build, list<paire>, list<p3D>> build;
6
7 typedef task<Slice, none_t, vector<int>> slice;
8 typedef task<Merge, vector<list<p3D>>, none_t> merge;
9
10 typedef stage<acq, rectif, detect, match, build> steps;
11 typedef sequence<steps> algo;
12
13 typedef scm<slice, worker<algo, N>, merge> app;
14 typedef application<app> recon3d;

```

On notera la simplicité de la description du schéma de parallélisation via le squelette **SCM** et comment le squelette **Sequence** permet de composer les différentes parties de notre algorithme.

6.2.2.3 Estimation des performances

À partir de ce schéma de parallélisation et du modèle de performance présenté au chapitre 2, on peut estimer *a priori* le gain que l’on peut attendre de cette implantation. Le tableau 6.4 présente les valeurs de gain maximal calculé via la formule proposée par Leo Chin Sim [159]. Pour effectuer ce calcul, nous avons repris les pourcentages relatifs des différentes étapes (tableau 6.2) et une évaluation purement SIMD des étapes de détection, mise en correspondance et triangulation.

	N=1	N=2	N=3	N=6	N=12	N=14
500	4.1	8.3	12.4	24.8	49.6	57.9
1000	4.5	8.9	13.4	26.8	53.6	62.5
2000	4.6	9.3	13.9	27.9	55.66	64.4

TAB. 6.4 – Modélisation des gains de l’application de reconstruction 3D

Si l’on se donne pour objectif d’atteindre une fréquence de 25 images par secondes, les gains nécessaires dans chaque cas de figure sont respectivement de 16

pour 500 points, 25 pour 1000 points et 44 pour 2000 points. Ces gains sont alors théoriquement atteints avec respectivement 6 et 12 nœuds, soit 12 à 24 processeurs.

6.2.3 Résultats

Les tableaux suivants présentent les mesures de performances pour l’application de reconstruction 3D parallèle. Les accélérations sont données pour une exécution sur une machine en mode monoprocesseur SIMD, une machine en mode biprocesseur scalaire, en mode MIMD seulement et sur l’ensemble du *cluster*. Les tests sont effectués pour 500 (tableau 6.8), 1000 (tableau 6.9) et 2000 points détectés (tableau 6.10) à la sortie de l’étape de détection et pour un nombre de noeuds allant de 1 à 14, soit 2 à 28 processeurs. Les temps mesurés sont moyennés sur 1000 itérations de l’algorithme.

<i>P</i>	Rectif.	Harris	Appar.	Triang.	Total	Accélération
500	243.4ms	95.7ms	97.6ms	33.8ms	470.5ms	1.4
1000	246.0ms	130.4ms	190.2ms	62.1ms	628.7ms	1.6
2000	244.3ms	151.2ms	438.6ms	139.3ms	973.4ms	1.8

TAB. 6.5 – Temps d’exécution de la reconstruction 3D en mode SIMD.

<i>P</i>	Rectif.	Harris	Appar.	Triang.	Total	Accélération
500	124.9ms	106.8ms	82.5ms	57.1	371.3ms	1.7
1000	126.2ms	145.6ms	198.8ms	105.9ms	576.5ms	1.7
2000	125.3ms	168.9ms	536.6ms	226.3ms	1057.1ms	1.7

TAB. 6.6 – Temps d’exécution de la reconstruction 3D en mode SMP.

<i>P</i>	N=1	N=2	N=3	N=6	N=12	N=14
500	1	1.9	2.8	5.4	9	10.2
1000	1	1.9	2.7	5.1	8.2	9.3
2000	1	1.9	2.5	5.8	10.7	13.8

TAB. 6.7 – Accélération de la reconstruction 3D en mode MIMD.

	N=1	N=2	N=3	N=6	N=12	N=14
Rectification	124.9ms	62.4ms	41.8ms	20.8ms	10.4ms	8.9ms
Détection	53.1ms	26.7ms	19.1ms	9.6ms	5.1ms	3.8ms
Mise en Corr.	72.1ms	39.5ms	23.1ms	11.1ms	5.2ms	4.3ms
Triangulation	21.4ms	9.6ms	6.4ms	2.6ms	1.9ms	1.4ms
Communication	0ms	4.8ms	5.6ms	6.4ms	7.6ms	8.2ms
Total	271.5ms	143ms	96ms	50.5ms	30.2ms	26.6ms
Accélération	2.4	4.56	6.8	12.9	21.6	24.5
Ecart au modèle	-41.5%	-45.1%	-45.1%	-47.9%	-56.5%	-57.7%

TAB. 6.8 – Temps d'exécution de la reconstruction 3D - 500 points

	N=1	N=2	N=3	N=6	N=12	N=14
Rectification	126.2ms	62.4ms	41.8ms	20.8ms	10.4ms	8.9ms
Détection	53.1ms	26.7ms	19.1ms	9.6ms	5.1ms	3.8ms
Mise en Corr.	124.1ms	64.8ms	44.3ms	22.3ms	11.2ms	8.8ms
Triangulation	36.4ms	19.0ms	11.4ms	4.6ms	2.9ms	2.4ms
Communication	0ms	6.6ms	8.2ms	9.2ms	12.1ms	12.9ms
Total	339.8ms	179.5ms	124.8ms	66.5ms	41.7ms	36.8ms
Accélération	2.9	5.5	7.9	14.9	23.7	26.9
Ecart au modèle	-48.9%	-38.2%	-41.1%	-44.4%	-29.5%	-56.9%

TAB. 6.9 – Temps d'exécution de la reconstruction 3D - 1000 points

	N=1	N=2	N=3	N=6	N=12	N=14
Rectification	125.3ms	62.4ms	41.8ms	20.8ms	10.4ms	8.9ms
Détection	53.1ms	26.7ms	19.1ms	9.6ms	5.1ms	3.8ms
Mise en Corr.	273.9ms	150.1ms	122.3ms	45.3ms	20.1ms	11.7ms
Triangulation	86.1ms	35.6ms	23.9ms	8.2ms	5.2ms	4.7ms
Communication	0ms	6.3ms	7.6ms	8.2ms	9.4ms	9.9ms
Total	538.4ms	281.1ms	214.7ms	92.1ms	50.2ms	39ms
Accélération	3.3	6.3	8.3	19.3	35.4	45.5
Ecart au modèle	-28.3%	-32.3%	-40.3%	-30.8%	-36.5%	-29.4%

TAB. 6.10 – Temps d'exécution de la reconstruction 3D - 2000 points

Plusieurs points apparaissent dans l'analyse de ces résultats. En terme de performance, les accélérations fournies par cette implémentation permettent une exécution temps réel en utilisant seulement une dizaine de noeuds pour 500 à 1000 points. On note aussi que les temps de communication restent très acceptables. Quel que soit le nombres de points et de noeuds, ces temps représentent au plus 33% du temps d'exécution total. L'architecture de multi-diffusion *FireWire* mise en œuvre montre de nouveau sa pertinence.

On peut néanmoins chercher à déterminer quelles sont les causes de l'écart non négligeable — entre 40% et 60% — entre les performances prédictes et celles effectivement mesurées. On montre que globalement, le gain fournit par pThread reste plus proche de 1.5 que de 2. Ainsi, une amélioration notable des gains serait envisageable en fusionnant les étapes de rectification, détection et mise en correspondance en une seule fonction qui ne nécessiterait qu'une unique création de contexte pThread. Une autre source de perte provient du fait que le gain fourni par ALTIVec au sein de l'étape de mise en correspondance est en partie perdu dans une série de recopies des zones de corrélation dans des espaces mémoires alignés.

L'autre point important est que pour de grandes quantités de points détectés, le gain augmente sensiblement. Pour de très grands nombre de points, ce gain se rapproche de celui prédit par le modèle de performance. Or, compte tenu du comportement du détecteur de point d'intérêt, le nombre de points correctement appariés et triangulés n'augmente que peu. En effet, les points d'intérêt supplémentaires ne sont pas suffisamment significatifs pour que leur appariement puisse se dérouler correctement. On atteint alors ici une limite plus algorithmique que programmatique.

6.3 Détection et suivi 3D temps réel de piétons

Le suivi d'objets est une problématique de vision artificielle dont les ramifications complexes ont donné naissance à de nombreux projets et méthodes. Le but de ces applications et de déterminer de manière précise la position et/ou la vitesse d'un objet quelconque au sein d'une séquence d'images. Parmi les différentes méthodes, on distingue les approches basées sur des modèles pré-établis de l'objet considéré et celles basées sur un apprentissage des objets à suivre. Nous proposons une méthode basée sur l'apprentissage [68] pour permettre le suivi de la trajectoire tridimensionnelle d'une personne en utilisant une formalisation probabiliste du problème.

6.3.1 Approche probabiliste du suivi d'objets

Dans cette approche probabiliste, les problèmes de suivi et de détection d'objets dans une séquence d'image se résument à rechercher la densité de probabilité *a posteriori* $P(\mathbf{X}_t | \mathbf{Z}_{1:t})$ à partir de la densité de probabilité $P(\mathbf{Z}_{1:t} | \mathbf{X}_t)$; où \mathbf{X}_t représente le vecteur d'états composé des paramètres du modèle de l'objet que l'on tente de suivre et $\mathbf{Z}_{1:t} = \mathbf{Z}_1, \dots, \mathbf{Z}_t$ est le vecteur d'observation qui regroupe l'historique des mesures effectuées précédemment. Ce processus récursif mets en oeuvre deux étapes :

- **Une étape de prédiction** dans laquelle on évalue $P(\mathbf{X}_t | \mathbf{Z}_{1:t-1})$ à partir de $P(\mathbf{X}_{t-1} | \mathbf{Z}_{1:t-1})$ et de la densité de transitions $P(\mathbf{X}_t | \mathbf{X}_{t-1})$. Pour ce faire, on utilise classiquement l'équation de prédiction de Chapman-Kolmogorov :

$$P(\mathbf{X}_n | \mathbf{Z}_{1:n}) = \int P(\mathbf{X}_n | \mathbf{X}_{n-1}) P(\mathbf{X}_{n-1} | \mathbf{Z}_{1:n-1}) d\mathbf{X}_{n-1}$$

- **Une phase de mesure et de mise à jour** dans laquelle l'utilisation de la règle de Bayes permet de mettre à jour la valeur de $P(\mathbf{X}_t | \mathbf{Z}_t)$ en fonction de $P(\mathbf{Z}_t | \mathbf{X}_t)$.

$$P(\mathbf{X}_n | \mathbf{Z}_{1:n}) = \frac{P(\mathbf{Z}_n | \mathbf{X}_n) P(\mathbf{X}_n | \mathbf{Z}_{1:n-1})}{P(\mathbf{Z}_n | \mathbf{Z}_{1:n-1})}$$

Avec :

$$P(\mathbf{Z}_n | \mathbf{Z}_{1:n-1}) = \int P(\mathbf{Z}_n | \mathbf{X}_n) P(\mathbf{X}_n | \mathbf{Z}_{1:n-1}) d\mathbf{X}_n$$

En théorie, ces équations récurrentes forment la base d'une estimation de Bayes optimale. Malheureusement, le calcul de ces intégrales est en général impossible analytiquement. Il faut alors avoir recours soit à des formes paramé-

triques des densités de probabilités, comme le filtre de Kalman [108] ou sa version étendue, soit à des techniques numériques comme, par exemple, les méthodes dites de Monte-Carlo, basées sur des tirages aléatoires.

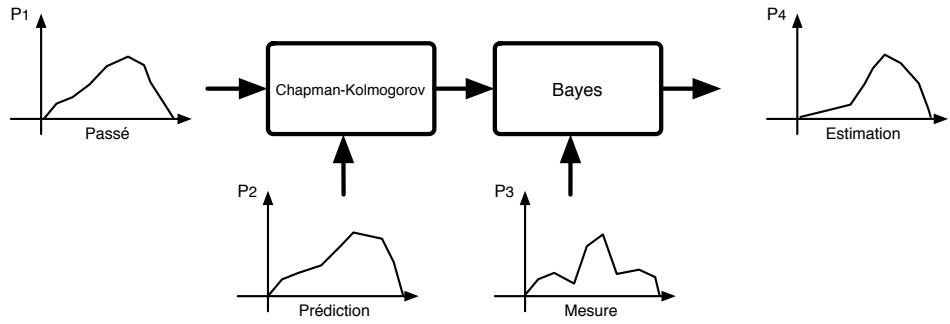


FIG. 6.11 – Approche probabiliste du suivi d’objet

6.3.2 Le filtre à particules

Le filtre à particules [100, 20, 177] est une méthode de Monte-Carlo séquentielle visant à estimer l'état d'un système à variables cachées modélisé par un processus Markovien. Le principe général du filtre à particules réside dans l'estimation de la densité de probabilité *a posteriori* $P(\mathbf{X}_t | \mathbf{Z}_{1:t})$ via un ensemble d'échantillons pondérés — ou particules — $\{(\mathbf{X}_t^n, \pi_t^n) : n = 1, \dots, N\}$ et une fonction d'observation $P(\mathbf{Z}_n | \mathbf{X}_n)$. La figure 6.12 décrit le pseudo-code du filtre à particules dans une application de suivi visuel tel que présenté dans [100].

1. **Initialiser** $\{(\mathbf{X}_0^n, \pi_0^n)\}_{n=1}^N$ à partir de la distribution *a priori* \mathbf{X}_0
2. **Pour** $t > 0$
 - (a) **Ré-échantillonner** $\{(\mathbf{X}_{t-1}^n, \pi_{t-1}^n)\}_{n=1}^N$ pour obtenir $\{(\mathbf{X}'_t^n, 1/N)\}_{n=1}^N$
 - (b) **Prédire** en générant $\mathbf{X}_t^n \sim P(\mathbf{X}_t | \mathbf{X}_{t-1} = \mathbf{X}'_t)$ pour obtenir $\{(\mathbf{X}_t^n, 1/N)\}_{n=1}^N$.
 - (c) **Pondérer**, en évaluant $\pi_t^n \propto P(\mathbf{Z}_t | \mathbf{X}_t = \mathbf{X}_t^n)$ pour obtenir $\{(\mathbf{X}_t^n, \pi_t^n)\}_{n=1}^N$ tel que $\sum_{n=1}^N \pi_t^n = 1$
 - (d) **Estimer** $\hat{\mathbf{X}}_t \doteq \sum_{n=1}^N \pi_t^n \mathbf{X}_t^n$

FIG. 6.12 – L’algorithme CONDENSATION [100]

La mise en œuvre de cet algorithme est relativement simple et ne nécessite que la définition d'un modèle de l'état de l'objet suivi, d'un modèle décrivant son évolution, d'une fonction d'observation et une stratégie d'initialisation.

6.3.2.1 Modèle d'état et d'évolution

Considérons une personne positionnée au temps t dans le repère du monde (figure 6.13). Le plan du sol est donnée par le plan (Oxy) dans lequel, cette personne est repérée par sa position 3D \mathbf{P}_t et sa hauteur h par rapport à son point de contact avec le sol. Le modèle d'état est alors défini comme étant :

$$\mathbf{X}_t = \begin{pmatrix} x_t \\ y_t \\ z_t \\ h_t \end{pmatrix}$$

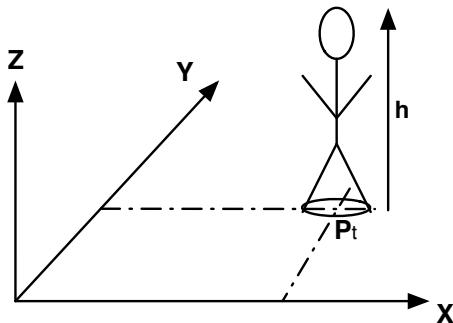


FIG. 6.13 – Modélisation de la position d'un piéton

Le modèle d'évolution — c'est à dire la relation existant entre l'état \mathbf{X}_t et \mathbf{X}_{t+1} — est ensuite donnée par la relation :

$$\mathbf{X}_{t+1} = \mathbf{A}\mathbf{X}_t + \mathbf{B}\mathbf{v}_t$$

où \mathbf{v}_t est un vecteur de bruit suivant une loi normale d'écart-type Σ . Les matrices \mathbf{A} , \mathbf{B} et Σ sont ensuite estimées à partir d'une échantillon d'image dans lesquelles la position du modèle suivi est connue. Dans notre cas, on obtient :

$$\mathbf{A} = \mathbf{B} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \sigma_x & 0 & 0 & 0 \\ 0 & \sigma_y & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_h \end{pmatrix}$$

Où $(\sigma_x, \sigma_y, \sigma_h)$ sont les écarts types des bruits gaussiens appliqués sur \mathbf{X}_t . Aucun bruit n'est appliqué sur z ; ce qui signifie que l'on fait l'hypothèse que la personne est en contact avec un sol plan. La valeur de z doit être néanmoins modélisée car la position de ce plan du sol est *a priori* inconnue.

6.3.2.2 Fonction d'observation

La fonction d'observation va permettre d'évaluer la vraisemblance d'une particule \mathbf{X}_t par rapport aux mesures. Dans notre cas, cette fonction passe par l'utilisation d'un **classifieur**, c'est à dire une fonction capable de fournir un score de vraisemblance pour une particule en fonction des données images disponibles. Comment évaluer la probabilité que l'hypothèse de localisation fournie par une particule corresponde à la position effective d'une personne dans l'image ? Parmi les méthodes récentes utilisées pour détecter des piétons, on peut citer des méthodes d'apprentissages comme les *Support Vector Machine* [34] ou ADABOOST [82]. Dans [37], on montre que l'on peut utiliser indifféremment ces deux méthodes. Nous avons donc choisi d'utiliser ADABOOST qui est une méthode basée sur le *boosting* [154, 178, 138]. Le principe du *boosting* est de combiner les résultats de plusieurs classificateurs faibles afin de créer un classificateur composite dont le pouvoir discriminant est supérieur à celui de chacun des classificateurs faibles. Par itérations successives, la connaissance d'un classificateur faible capable de reconnaître 2 classes d'objets au moins aussi bien que le hasard ne le ferait (c'est-à-dire qu'il ne se trompe pas plus d'une fois sur deux en moyenne) est pondérée par la qualité de sa classification et ajoutée au classificateur composite. Plus un classificateur faible classe bien, plus il sera considéré de manière importante à l'itération suivante.

Considérons une base d'apprentissage $\{(x_1, y_1), \dots, (x_m, y_m)\}$ où les x_i sont des éléments d'un domaine X et où y_i les identifiant de la classe associée à x_i tel que $\forall y_i, y_i \in \{-1, +1\}$. On se donne ensuite $\mathcal{H} = \{h_1, \dots, h_t\}$ un ensemble de classificateurs. ADABOOST va alors procéder de la manière suivante :

- Initialisation de la distribution D_1 telle que $D_1(i) = \frac{1}{m}$.
- Pour t allant de 1 à T :
 - Trouver le classificateur $h_t \in \mathcal{H}$ qui minimise l'erreur vis à vis de D_t .

$$h_t = \underset{h_j \in \mathcal{H}}{\operatorname{argmin}} \varepsilon_j, \quad \text{avec } \varepsilon_j = \sum_{i=1}^m D_t(i) [y_i \neq h_j(x_i)] < 0.5$$

$$- \text{ Évaluer } \alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right).$$

- Mettre à jour :

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

où Z_t est un facteur de normalisation qui garantit que D_{t+1} reste une distribution.

- Construire le nouveau classifieur $H(x) = \text{sign}(\sum_{t=1}^T \alpha_t h_t(x))$

L'équation de mise à jour de D_t est construite de manière à ce qu'après avoir sélectionné un classifieur optimal h_t pour la distribution, l'échantillon x_i que h_t a correctement reconnu se voit affecter un poids plus faible. Ainsi, à l'itération suivante, lorsque l'algorithme va rechercher un nouveau classifieur optimal pour D_{t+1} , il sélectionnera un classifieur h_{t+1} qui identifiera mieux les exemples délaissés par h_t . Dans notre cas, nous utilisons des classificateurs naïfs basés sur une description de l'image par des ondelettes de Haar (figure 6.14) qui permettent de détecter certains nombres de caractéristiques du piéton comme son aspect vertical et symétrique. En outre, comme nous l'avons vu pour l'application de stabilisation d'images, ces filtres peuvent se calculer très rapidement à partir d'une image intégrale.

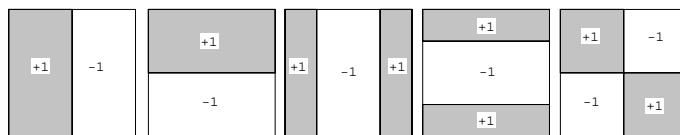


FIG. 6.14 – Détecteur à ondelettes de Haar

Pour effectuer l'apprentissage hors-ligne, nous utilisons une base de 4000 images de piétons utilisée dans des travaux préliminaires [143, 37] (figure 6.15). Les éléments de cette base d'apprentissage sont des images 64x128 dans lesquelles se trouve centré un piéton de dos ou de face.



FIG. 6.15 – Échantillon de la base d'apprentissage de piéton

La base d'images des non-piétons est quant à elle constituée de 2000 images de tailles variables et représentent divers objets et textures d'intérieur et d'extérieur.

À partir du classifieur composite construit par l'algorithme ADABOOST en utilisant ces bases d'apprentissages et ces classifieurs, l'algorithme utilisé pour évaluer le poids d'une particule consiste, pour chaque particule \mathbf{X}_t^n , à calculer la projection des points $O_t^n = (x_t, y_t, z_t)$ et $H_t^n = (x_t, y_t, z - t + h_t)$ dans les repères images de chaque caméra. On extrait ensuite une image qui correspond à une boîte englobant O_t^n et H_t^n dont le ratio hauteur/largeur vaut 2 (fig. 6.16).

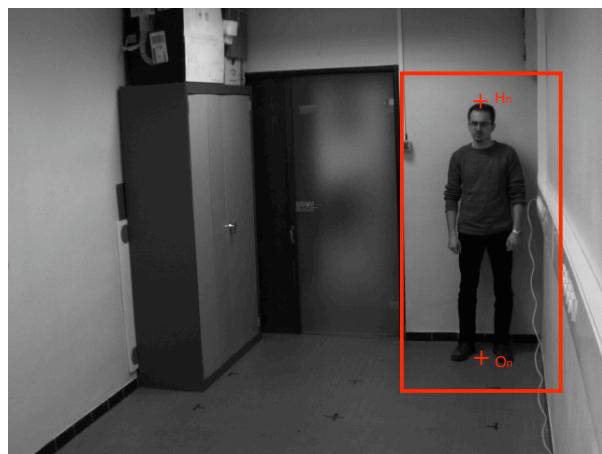


FIG. 6.16 – Boîte englobante d'un piéton

Ce découpage correspond exactement à la forme générale des images de piétons utilisées dans la phase d'apprentissage hors-ligne. Cela implique que la zone de suivi effectif est restreinte à la zone où cette boîte englobante est visible dans l'image. Ainsi, si un piéton se trouve dans la partie inférieure de l'image, la boîte englobante calculée sort de cette dernière et se voit affecter un poids nul. Ces images sont alors ramenées par sous-échantillonage à une taille de 64x128 sur lesquelles on effectue une mesure en utilisant le classifieur composite créé par l'algorithme ADABOOST en calculant les images intégrales nécessaires à l'évaluation des ondelettes de Haar. On obtient alors un score de vraisemblance pour la projection de la particule dans l'image droite et un score de vraisemblance pour la projection de la particule dans l'image gauche. On se ramène à un unique score en prenant la moyenne de ces deux scores. D'autres fonctions ont été envisagées pour effectuer cette mesure comme la moyenne géométrique ou le minimum des

deux scores. Empiriquement, il s'avère que la moyenne donne des résultats plus stables et moins sensibles aux cas limites dans lesquels un des deux scores est extrêmement petit par rapport à son homologue.

6.3.2.3 Etape d'initialisation

La phase d'initialisation permet de générer la distribution de particules initiale. Notre stratégie consiste à discréteriser la zone de détection. Ainsi, la zone de suivi de 3m par 2m est décomposée en section de 1cm par 1cm, soit 60000 points. Pour chaque point, on applique notre fonction de mesure. Une fois ces mesures effectuées, on ne conserve que les 1000 meilleurs scores, afin de restreindre la zone de suivi et le nombre de particules à manipuler.

6.3.3 Implantation séquentielle

L'algorithme séquentiel du suivi par filtrage particulaire peut être grossièrement décrit par une succession d'étapes reprenant l'algorithme CONDENSATION présenté ci-dessus :

- une étape de génération qui consiste à évaluer \mathbf{X}_t^n à partir de \mathbf{X}_{t-1}^n en utilisant le modèle d'évolution;
- une étape de pré-traitement des images qui consiste à calculer les images intégrales qui seront utilisées par la fonction de mesure;
- une étape de mesure qui évalue une série d'ondelettes de Haar et utilise la base d'apprentissage issue de l'algorithme ADABOOST pour évaluer le score de vraisemblance de chaque particule;
- une étape de mise à jour qui évalue les poids \mathbf{Z}_t^n de la distribution \mathbf{X}_t^n ;
- une étape d'estimation qui calcule ici la position de la personne suivie en évaluant la moyenne pondérée de la distribution de particules;
- une étape d'échantillonnage qui modifie \mathbf{X}_t^n en répliquant en son sein les particules en fonction de leur poids.

Le tableau 6.11 présente alors les temps d'exécutions de cette implantation pour un nombre de particules variant de 100 à 10000. On se place dans l'hypothèse du régime permanent et on ne tient pas compte du temps nécessaire à l'étape d'initialisation. Ces résultats sont obtenus sur une machine POWER PC G5 en mode mono-processeur et sans utiliser l'extension ALTIVEC.

N_{part}	Temps d'exécution	Débit
100	61,1 ms	16,38 images/sec
500	299,5 ms	3,34 images/sec
1000	672,3 ms	1,49 images/sec
2000	1986,2 ms	0,50 images/sec
5000	11358,4 ms	0,09 images/sec
10000	21142,7 ms	0,05 images/sec

TAB. 6.11 – Performances séquentielles du suivi de personne par filtrage particulaire

L'algorithme séquentiel est relativement performant. Pour des distributions de particules de moins de 200 éléments, son exécution se déroule en temps réel. Plusieurs points viennent néanmoins plaider en faveur d'une implantation parallèle de cet algorithme. Tout d'abord, la qualité du suivi en termes de précision et de stabilité est fortement liée au nombre de particules utilisées. Dans notre cas, il faut utiliser 500 à 1000 particules pour obtenir un suivi stable. L'autre point important est que l'algorithme de filtrage particulaire nécessite l'utilisation d'un nombre de particules qui croît exponentiellement avec la taille du vecteur d'état. Ainsi, si l'on désire étendre cet algorithme au suivi de plusieurs personnes avec une approche intégrant dans le vecteur d'état les N personnes à détecter et à suivre [101], il faudrait de l'ordre de 1000 particules pour suivre seulement deux personnes.

6.3.4 Implantation parallèle

Notre objectif est donc de fournir une implantation parallèle de l'algorithme de suivi par filtrage particulaire. Plusieurs travaux ont été menés afin d'implanter un tel algorithme sur des architectures parallèles de types FPGA ou en se restreignant à la parallélisation de l'étape de rééchantillonnage. En ce sens, il n'existe pas à notre connaissance d'implantation parallèle complète de ce type d'algorithme sur une machine de type *cluster*. Le but est de permettre l'exécution à la fréquence d'acquisition vidéo de l'algorithme pour un nombre de particules de l'ordre de 1000 à 2000 en proposant une implantation parallèle pertinente.

6.3.4.1 Analyse du problème séquentiel

L'analyse de l'implantation séquentielle du suivi par filtrage particulaire est résumée dans le tableau 6.12.

<i>N</i>	Génération	Pré-traitement	Mesure	Mise à jour	Estimation	Echantillonnage
100	0,28%	17,23%	81,39%	0,05%	0,25%	0,8%
500	0,13%	3,51%	95,23%	0,02%	0,12%	0,98%
1000	0,12%	1,56%	96,51%	0,02%	0,11%	1,68%
2000	0,08%	0,53%	97,13%	0,01%	0,07%	2,17%
5000	0,04%	0,09%	97,47%	0,01%	0,04%	2,35%
10000	0,05%	0,05%	94,57%	0,01%	0,05%	5,28%

TAB. 6.12 – Importance relative des étapes de l'algorithme de suivi

Elle met en évidence un certain nombre de points. Tout d'abord, les étapes de génération des particules, de mise à jour et d'estimation ne représente que 0.5% du temps d'exécution total. Ensuite, l'étape de **mesure** représente entre 70% et 90% du temps total de l'exécution de l'application. Elle concentre en effet un grand nombre de traitements — principalement les projections des particules dans les images et le calcul des scores de vraisemblance. De même, l'étape **d'échantillonnage** est une étape dont le comportement ne devient critique que pour de grands ensembles de particule car son temps d'exécution est fonction du carré du nombre de particules. Par contre, l'étape de **pré-traitement**, qui consiste à pré-calculer les images intégrales nécessaire à l'étape de mesure, représente une portion de moins en moins importante du temps d'exécution. Elle ne dépend en effet que de la taille des images.

6.3.4.2 Proposition d'implantation

À la lumière de cette analyse, nous proposons le schéma d'implantation parallèle suivant :

- L'étape de mesure va bénéficier d'une parallélisation intensive. Les mesures à effectuer pour chaque particule vont être distribuées sur l'ensemble des nœuds disponibles. Ensuite, sur chaque nœud, la mesure en elle-même va utiliser les deux processeurs ainsi que l'unité ALTIVec afin d'accélérer les calculs géométriques nécessaires. L'algorithme de classification ne profitera quant à lui que du gain fourni par ALTIVec;
- L'étape de pré-traitement va s'effectuer de manière simultanée sur chaque nœud. À chaque itération de l'algorithme, tous les nœuds ont besoin d'accéder à l'image intégrale complète pour effectuer leur mesure. Cette étape est donc répliquée sur tous les nœuds et ne bénéficie pas d'autre forme de

parallélisme;

- Une fois les mesures, la mise à jour et l'estimation effectuées, l'étape de re-échantillonnage va nécessiter un parallélisation *ad hoc* qui consiste à diffuser, via un *broadcast*, l'ensemble du nouveau tableau de particules à tous les nœuds. Chaque nœud effectue le calcul d'une portion d'un vecteur de permutation qui, une fois récupéré sur le nœud maître va permettre de reconstruire le tableau re-échantillonné des particules.

Le tableau 6.13 résume les choix de parallélisation effectués qui donne le graphe de processus communicant associé.

Étapes	SIMD	SMP	MIMD
Génération	—	—	—
Pré-traitements	—	—	✓
Mesure	✓	✓	✓
Mise à jour	—	—	—
Estimation	—	—	—
Echantillonage	—	—	✓

TAB. 6.13 – Parallélisation de l'algorithme de suivi

Ce schéma de parallélisation est résumé dans la figure 6.17.

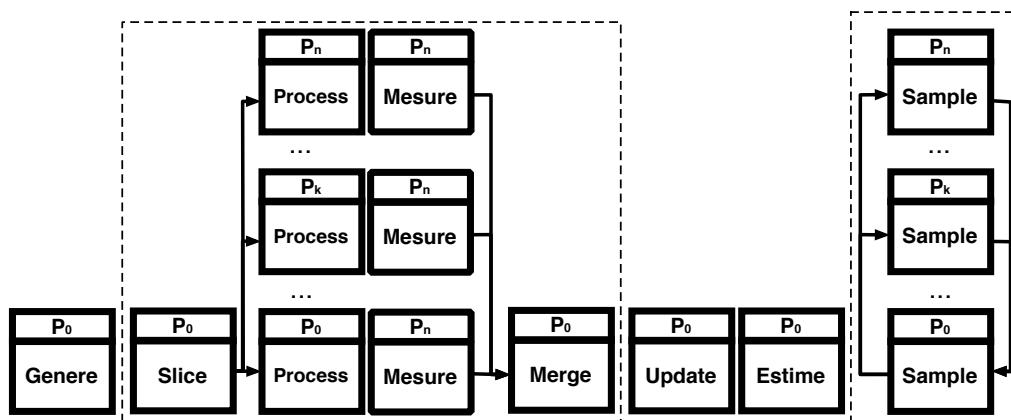


FIG. 6.17 – Graphe de processus communiquant pour l'application de suivi

On reconnaît ici l'utilisation d'un squelette **Sequence** contenant les fonctions effectives de l'algorithme, un squelette **SCM** pour la parallélisation de la fonction de mesure et de pré-traitement et un squelette **Pardo** pour la fonction d'échantillonnage à base de *broadcast*. Son expression sous forme de squelette se représente alors comme sur la figure 6.18.

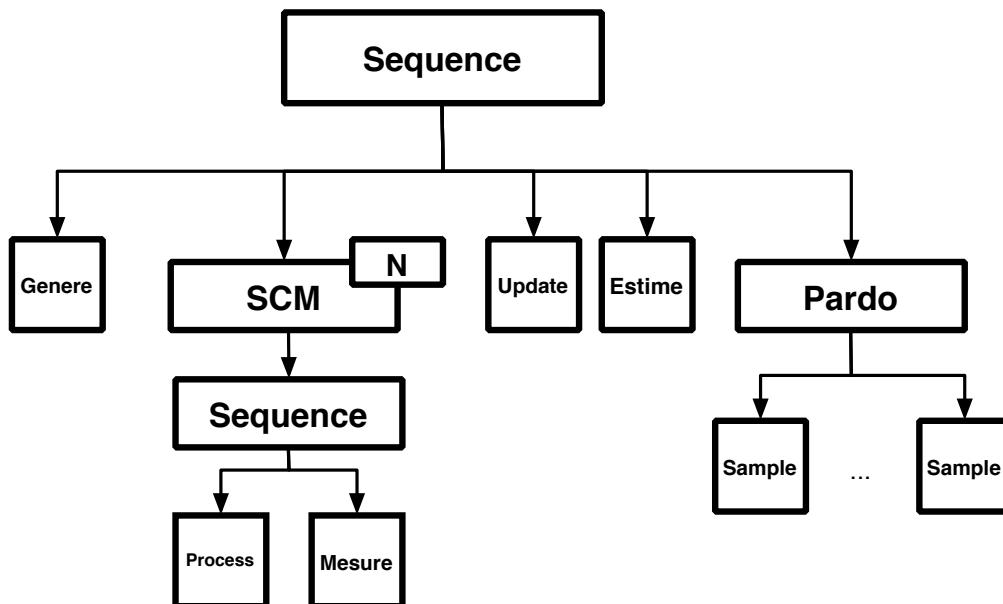


FIG. 6.18 – Schéma d'imbrication des squelettes de l'application de suivi

Le code QUAFF de cette application se décompose en six parties :

- l'intégration des fonctions utilisateurs via la construction task.

```

1  typedef task<Generate,none_t,x_t>           gen;
2  typedef task<ImgProcess,x_t,Data>           process;
3  typedef task<Measure,Data,w_t>             mesure;
4  typedef task<Measure,w_t,p_t>              update;
5  typedef task<Measure,p_t,p_t>              estim;
6  typedef task<BResample,p_t,none_t>         bresample;
7  typedef task<Resample,none_t,none_t>        dresample;
  
```

Dans ces définitions, les types **p_t**, **w_t** et **Data** correspondent respectivement aux types utilisateurs encapsulant le tableau des particules, le tableau des poids des particules et une structure contenant la paire d'image pré-

traitée nécessaire à la fonction de mesure;

- l'intégration des fonctions de diffusion et récupération du squelette **SCM**.

```
1  typedef task<Slice, part_t, vector<p_t>> slice;
2  typedef task<Merge, list<vector<w_t>>, w_t> merge;
```

- la définition du squelette **Séquence** composé de la fonction de pré-traitement et de mesure.

```
1  typedef sequence<stage<process, mesure>> algo;
```

- la définition du squelette **SCM** associé à cette **Séquence**.

```
1  typedef scm<slice, worker<algo, N>, merge> eval;
```

- la définition du squelette **Pardo** décrivant l'étape d'échantillonnage parallèle.

```
1  typedef stage<bresample, repeat<dresample, N-1>> stg;
2  typedef pardo<stg> sample;
```

Ici, la méta-fonction `repeat<F, N>` est un raccourci de syntaxe qui exprime la répétition d'une seule et même fonction `F` sur `N` nœuds. Elle répond de manière *ad hoc* au problème de lisibilité du squelette **Pardo**. Les fonctions contenues dans ce squelette correspondent à l'étape de *broadcast* — `bresample` — et aux étapes de calculs effectifs — `dresample`. On notera la différence entre les types d'entrées/sorties de leurs déclarations respectives qui reflètent bien le fait que la stratégie de communications est gérée implicitement par les fonctions contenues dans **Pardo**.

- La déclaration de la **Séquence** principale et de l'application finale.

```
1  typedef stage<gen, eval, update, estim, sample> track;
2  typedef sequence<track> tracker;
3  typedef application<tracker> app;
```

Le code complet de l'application est donné dans le listing 6.1.

Listing 6.2 – Implantation QUAFF de l’application de suivi par filtrage particulaire

```

1  typedef task<Generate, none_t, x_t>                                gen;
2  typedef task<ImgProcess, x_t, Data>                                 process;
3  typedef task<Measure, Data, w_t>                                mesure;
4  typedef task<Measure, w_t, p_t>                                update;
5  typedef task<Measure, p_t, p_t>                                estim;
6  typedef task<BResample, p_t, none_t>                            bresample;
7  typedef task<Resample, none_t, none_t>                           dresample;
8
9  typedef task<Slice, part_t, vector<p_t>>                      slice;
10 typedef task<Merge, list<vector<w_t>>, w_t>                  merge;
11
12 typedef sequence<stage<process, mesure>>                     algo;
13 typedef scm<slice, worker<algo, N>, merge>                   eval;
14 typedef stage<bresample, repeat<dresample, N-1>>           stg;
15 typedef pardo<stg>                                         sample;
16
17 typedef stage<gen, eval, update, estim, sample>                track;
18 typedef sequence<track>                                       tracker;
19 typedef application<tracker>                                    app;

```

6.3.4.3 Évaluation des performances

Comme pour l’algorithme de reconstruction 3D, nous allons estimer le gain que l’on peut attendre de cette implantation. Le tableau 6.14 présente les valeurs de gain maximal calculé par notre modèle de performance en utilisant la répartition du temps de calcul mise en avant précédemment.

N	$n_P=100$	$n_P=500$	$n_P=1000$	$n_P=2000$	$n_P=5000$	$n_P=10000$
1	6,33	7,71	7,79	7,83	7,85	7,67
2	12,65	15,42	15,57	15,65	15,69	15,35
3	18,97	23,13	23,36	23,47	23,54	23,02
6	37,94	46,25	46,72	46,95	47,08	46,03
12	75,88	92,50	93,43	93,89	94,16	92,07
14	88,53	107,92	109,00	109,54	109,85	107,41

TAB. 6.14 – Estimation des accélérations de l’algorithme de suivi parallèle

Le gain attendu pour implantation est très élevé. Ceci provient principalement

du fait que l'étape subissant une parallélisation sur 3 niveaux — MIMD, SMP et SIMD — représente une large partie du temps d'exécution de l'application. Compte tenu de ces gains et des temps d'exécution séquentiels mesurés précédemment, l'exécution temps réel de cet algorithme est potentiellement possible avec seulement 2 à 8 nœuds (soit 4 à 16 processeurs) pour des jeux de particules de 500 à 2000 éléments. Pour des jeux de 5000 particules, l'accélération maximale estimée ne garantira qu'une exécution à 10 images par secondes pour 5000 particules et de seulement 3 images par secondes pour 10000 particules.

6.3.5 Résultats

Le tableau 6.15 présente les résultats temporels de l'algorithme de suivi parallèle pour des jeux de particules allant de 100 à 10000 éléments. Les résultats donnés sont : le temps de calcul effectif, le temps de communications, le temps d'exécution total, le nombre d'images par secondes et le gain par rapport à la version séquentielle. Ces résultats sont données pour des configurations comportant 1 à 14 nœuds — soit 2 à 28 processeurs. L'analyse de ces performances met en évidence plusieurs points :

- En terme de performances absolues, nous obtenons des temps d'exécutions compatibles avec l'exécution en temps réel vidéo — soit 20 à 30 images par secondes — pour des jeux de particules de 2000 éléments avec 28 processeurs. En terme de gain, cette exécution à la fréquence d'acquisition vidéo représente une accélération de l'ordre de 46. En outre, l'évolution de ce gain est linéaire avec le nombre de processeurs, ce qui indique que notre stratégie de parallélisation est très satisfaisante;
- pour les jeux de particules de faibles tailles — typiquement moins de 2000 particules — les temps d'exécutions obtenues sont tous compatibles avec l'exécution temps réel en bénéficiant de gain de l'ordre de 2 à 25 avec seulement 1 à 6 nœuds;
- pour des jeux de particules de plus grandes tailles, le temps d'exécution devient trop important pour que la cadence soit compatible avec nos contraintes temps réel. Pour 28 processeurs et 5000 particules, on atteint approximativement 10 images par secondes avec un gain de l'ordre de 100. On peut alors estimer que l'exécution temps réel dans ces conditions nécessiterait peut être l'ajout de 1 à 2 noeuds supplémentaires au *cluster*. Pour 10000 particules, nous atteignons à peine 4 images par secondes malgré un gain

de près de 83. Nous atteignons ici une limite en terme de temps d'exécution due au fait que la version séquentielle de cet algorithme est peu performante. Malgré un gain très élevé — et donc une parallélisation de qualité — nous ne pouvons atteindre un temps d'exécution satisfaisant.

La plate-forme BABYLON montre qu'elle permet de réaliser l'exécution temps réel d'un algorithme de suivi par filtrage particulaire pour des jeux de particules de tailles importantes. Les gains et les temps d'exécution obtenus correspondent bien à nos attentes et valident le choix de l'implantation parallèle de cette algorithme.

6.4 Conclusion

Nous avons implanté et exécuté **deux applications de vision artificielle complexes** sur la machine BABYLON. Ces deux applications — reconstruction 3D et suivi de personne par filtrage particulaire — ont été exécutées **à une cadence compatible avec le temps réel vidéo** — c'est-à-dire à une cadence de 20 à 30 images par seconde — tout en exhibant **des gains de l'ordre de 30 à 100 avec seulement 28 processeurs**, confirmant par la même la pertinence de l'architecture BABYLON. Ces tests ont en particulier mis en évidence deux points.

D'une part, BABYLON peut fournir des accélérations allant de 20 à 100 et permet donc de répondre au problème de l'exécution d'applications de vision artificielle complexes dans un contexte temps réel et ce sans nécessiter de dégradation substantielle des algorithmes mis en œuvre. D'autre part, l'utilisation de E.V.E. et QUAFF au sein d'applications réalistes est aisée et aboutit à des performances très satisfaisantes. À titre indicatif, le temps de développement de la version séquentielle de l'application de reconstruction 3D a été de quelques mois mais sa parallélisation via QUAFF n'a quant à elle pris que une à deux semaines. Pour l'application de suivi, nous sommes parti d'un code source MATLAB®. Son portage en C++ via E.V.E. et sa parallélisation via QUAFF ont nécessité deux semaines. Ces temps de développement confirme donc l'intérêt de tels outils de développement.

Ces résultats valident donc la pertinence et les performances de notre plate-forme de développement et des outils qui lui sont associés.

	$n_P=100$	$n_P=500$	$n_P=1000$	$n_P=2000$	$n_P=5000$	$n_P=10000$
$N = 1$						
Calcul	41.96ms	88.68ms	170.13ms	338.76ms	856.04ms	1796.93ms
Comm.	0.05ms	0.07ms	0.05ms	0.04ms	0.07ms	0.08ms
Total	42.01ms	88.75ms	170.18ms	338.80ms	856.11ms	1797.01ms
FPS	23.81	11.27	5.88	2.95	1.16	0.55
Gain	1.45	3.37	3.95	5.86	13.26	11.76
$N = 2$						
Calcul	25.05ms	47.85ms	90.82ms	174.21ms	451.78ms	960.81ms
Comm.	1.8ms	2.71ms	2.80ms	3.76ms	4.56ms	7.86ms
Total	26.85ms	50.56ms	93.62ms	177.97ms	456.34ms	968.67ms
FPS	37.24	19.78	10.68	5.62	2.19	1.03
Gain	2.27	5.92	7.19	11.16	24.89	21.82
$N = 3$						
Calcul	18.34ms	35.4ms	63.01ms	119.57ms	109.86ms	688.56ms
Comm.	2.76ms	2.83ms	3.81ms	4.66ms	9.71ms	13.32ms
Total	21.01ms	38.23ms	66.82ms	124.23ms	119.57ms	701.88ms
FPS	47.59	26.16	14.97	8.05	3.16	1.45
Gain	2.90	7.83	10.06	15.99	35.90	30.12
$N = 6$						
Calcul	13.60ms	22.89ms	33.03ms	57.79ms	165.04ms	390.98ms
Comm.	2.95ms	3.39ms	7.7ms	15.02ms	18.11ms	20.22ms
Total	16.55ms	26.28ms	40.73ms	72.81ms	183.15ms	411.20ms
FPS	60.42	38.05	24.55	13.73	5.46	2.43
Gain	3.68	11.39	16.50	27.23	62.01	51.42
$N = 12$						
Calcul	11.50ms	16.06ms	22.09ms	29.26ms	99.67ms	247.78ms
Comm.	2.93ms	3.50ms	10.14ms	14.62ms	16.81ms	23.99ms
Total	14.43ms	19.56ms	32.23ms	46.88ms	116.48ms	271.17ms
FPS	69.30	51.13	31.1	21.33	8.58	3.68
Gain	4.22	15.31	20.85	42.37	97.51	77.96
$N = 14$						
Calcul	10.46ms	15.53ms	15.30ms	29.97ms	85.83ms	227.15ms
Comm.	3.66ms	3.94ms	12.02ms	12.81ms	19.97ms	25.75ms
Total	14.12ms	19.47ms	27.32ms	42.78ms	105.80ms	252.90ms
FPS	70.84	51.35	36.61	23.36	9.45	3.95
Gain	4.31	15.38	24.60	46.43	107.35	83.60

TAB. 6.15 – Performances temporelles du suivi parallèle

Conclusion et perspectives

«*There is no problem so complicated that you can't find a very simple answer to it if you look at it right ... Or put it another way, the future of computer power is pure simplicity.*»
Douglas Adams, *The Salmon of Doubt*

Les travaux exposés dans ce manuscrit ont pour but de montrer qu'une architectures de type *cluster* peut répondre de manière efficace aux besoins de la vision artificielle temps réel. Cet objectif nous a amené à développer une solution basée sur une architecture matérielle et des outils de développement adaptés à cette problématique. Ce chapitre reprend, de manière synthétique, les apports de chacune de ces réalisations et présente quelques pistes de recherches ouvertes par ces dernières.

L'architecture BABYLON

Nous avons montré qu'il existait, au sein des diverses problématiques de la vision artificielle, un certain nombre d'applications dont l'utilisation dans un contexte dit «temps réel» était limitée par la complexité ou la quantité des calculs à effectuer. Pour répondre à ces besoins, nous avons montré que l'exploitation du parallélisme sur des architectures de type *cluster* fournissait une réponse qui était susceptible de répondre aux contraintes de ces applications.

L'objectif premier était donc de fournir une architecture parallèle de type *cluster* permettant l'implantation d'applications de vision artificielle limitée par les calculs afin de les exécuter dans un contexte temps réel. Après une courte étude, nous avons mis en évidence un problème du à **la latence due aux communications entre les nœuds du cluster et les capteurs images**. Notre problématique scientifique était donc double :

- proposer une architecture de *cluster* qui limite ces problèmes de latence;
- montrer que cette architecture est une réponse adaptée en terme de coût et de performances.

Nous avons donc défini une nouvelle architecture — BABYLON [72] — qui tire parti et étend les travaux proposés par Revenga [149] et Yoshimoto [91] s'appuyant sur un réseau de communication hybride dans lequel les transferts de données image sont réalisés sur un réseau compatible avec les capteurs images et les transferts provenant de l'applications sont effectués sur un réseau Ethernet classique. Par ailleurs, BABYLON utilise **trois niveaux de parallélisme** en son sein : un niveau de parallélisme MIMD, un niveau de parallélisme SMP grâce à la présence de deux processeurs sur chaque nœud et un niveau SIMD sous la forme d'une extension intra-processeur — l'extension ALTIVEC. Enfin, cette architecture a nécessité le développement d'un *driver* Firewire capable d'alimenter efficacement les nœuds de calcul du *cluster* : la bibliothèque C+FOX.

La mise en œuvre de cette architecture sur des applications de complexité réaliste a démontré que l'implantation de telles applications tirait pleinement parti de ces spécificités en éliminant une grande partie des temps de latence dus aux transferts des données images sur le réseau Ethernet. Elle démontre aussi que l'utilisation des **trois niveaux de parallélisme** permet de réduire le nombre de nœuds nécessaire pour atteindre des gains permettant d'exécuter ces applications de vision artificielles [67, 68] en temps réel. Ainsi, BABYLON fournit des **gains allant de 20 à 100 avec seulement quatorze machines**.

Les bibliothèques E.V.E. et QUAFF

La programmation d'application sur une telle architecture n'est pas un problème trivial. De nombreux outils et modèles ont été proposés pour permettre de faciliter cette tâche, tant pour la programmation MIMD que pour l'utilisation d'extension SIMD comme ALTIVEC. Les principaux défauts de ces outils résident dans le compromis qui doit souvent être trouvé entre leur expressivité et les performances du code qu'ils produisent. Comme pour l'architecture matérielle de BABYLON, nous avons cherché à définir des techniques permettant de proposer des outils de développement parallèle à la fois expressifs et efficaces.

La technique retenue utilise un aspect assez peu usité du langage C++. Au sein de ce langage, le mécanisme des *templates* — ou patrons — permet en effet

de mettre en œuvre l'équivalent d'un **système d'évaluation partielle de code**. Ce système permet de générer un code très proche d'un code C écrit et optimisé à la main tout en conservant une interface utilisateur très expressive. Nous proposons alors deux outils sous forme de bibliothèque C++ : E.V.E. [70, 71], qui gère le développement de code de calcul scientifique utilisant ALTIVEC, et QUAFF [67], qui implante en C++ un modèle de programmation à base de squelettes algorithmiques.

E.V.E. a montré que l'application des techniques d'évaluation partielle au problème du calcul numérique haute performance permet d'utiliser de manière transparente un grand nombre de techniques d'optimisations. Plus particulièrement, l'expression naturelle du parallélisme SIMD sous forme de tableaux numériques optimisables permet de limiter la réécriture d'algorithmes bas niveau afin de profiter des performances d'une extension SIMD comme ALTIVEC. Dans le cas de QUAFF, les techniques d'évaluation partielle ont permis de fournir une bibliothèque à la fois performante et fidèle à un modèle de programmation éprouvé.

Perspectives

À partir de ces résultats encourageants, plusieurs axes de recherches sont envisageables. En terme d'architecture, BABYLON est un modèle générique. Créer un nouveau *cluster* basé sur les mêmes principes mais utilisant des machines Intel sous Linux, par exemple, ou utilisant moins de noeuds est tout à fait envisageable. Cette générnicité de l'architecture est d'ailleurs supportée par la **portabilité des outils logiciels E.V.E. et QUAFF**. Compte tenu des performances de cette architecture hybride, on peut envisager de créer une version réduite de BABYLON comportant 4 ou 6 noeuds plus conventionnels et pouvant alors être embarquée (au sein d'un véhicule par exemple).

Une autre version de BABYLON peut aussi être envisagée en équipant les noeuds du *cluster* de récepteurs WiFi. Cette version de l'architecture nécessiterait bien évidemment une adaptation en terme du *driver* Firewire mais permettrait de mettre en place une **plate-forme de calcul déporté** capable de commander un robot mobile équipé d'une caméra par exemple. Ce robot transmettrait alors son flux vidéo via le réseau WiFi à la machine BABYLON qui effectuerait les calculs nécessaires à sa commande. Bien entendu, dans une telle configuration, le point important à prendre en compte est la latence et la faible bande passante d'un tel réseau qui posent de nouveaux problèmes technologiques à résoudre.

Au niveau logiciel, outre l'ajout progressif de nouvelles fonctionnalités, E.V.E. ouvre une nouvelle voie pour l'intégration efficace de diverses technologies multimédia. La structure interne de E.V.E. est en effet facilement configurable pour supporter d'autres extensions SIMD. Une version SSE/SSE2 est d'ores et déjà en cours de développement et l'expérience acquise dans ce portage nous conforte dans l'idée que la majorité des extensions de ce type peuvent être supportées par E.V.E.. Pour QUAFF, de nombreux travaux restent à mener pour parfaire l'optimisation du code généré pour homogénéiser la gestion des types définis par l'utilisateur ou l'intégration de code nécessitant des schémas de synchronisations complexes qui restent encore imparfaits.

L'utilisation conjointe de ces deux bibliothèques a aussi montré les limites de leurs modèles de programmation respectifs. La question se pose en particulier de définir une limite entre ces derniers. Si l'expression du parallélisme de contrôle et celle du parallélisme de données de grain fin sont naturellement distribués entre QUAFF et E.V.E. respectivement, quel est la meilleure manière d'exprimer le parallélisme de donnée à gros grains ? En effet, une extension de E.V.E. visant à doter les classes `array` et `view` de méthodes permettant leur distribution et leur réductions sur une machine MIMD via l'utilisation de MPI est parfaitement envisageable mais on disposerait alors de deux moyens d'expression du parallélisme de données à gros grain qui compliquerait alors la tâche du développeur. Une des pistes de développement consisterait alors à clarifier les parties gérées uniquement par QUAFF, celles gérées uniquement par E.V.E. et de déterminer une interface idoine pour le traitement des types de parallélismes exprimables sous plusieurs formes. Dans une optique similaire, l'utilisation du parallélisme SMP via pThread — qui s'est avéré être la partie la plus complexe des développements effectués — pose la question de la création d'une interface adéquate, soit par l'intégration au sein de E.V.E. — via l'adjonction de nouveaux marqueurs et l'écriture de méta-programme *ad hoc* — soit au sein d'une nouvelle bibliothèque qui utiliserait un modèle de programmation dédié.

Au-delà de ces considérations et plus généralement, l'opportunité de développer la plus flagrante de ces travaux est l'application des techniques dérivées de l'évaluation partielle aux problématiques d'implantation efficace de modèles haut-niveau. Malgré leurs différences fondamentales en termes d'interface, E.V.E. et QUAFF représentent en effet deux applications de ces mêmes techniques. Leur point commun réside dans l'analyse fine du problème et l'extraction de cette analyse d'un schéma d'évaluation statique. On peut alors se demander si des outils

du même genre ne pourraient pas être proposés pour fournir une implantation efficace en C++ d'outils dédiés à des problèmes comme la composition d'opérateurs de traitements d'images de haut-niveau [156] ou l'expression directe en C++ du parallélisme basé sur d'autres modèles que les squelettes. En tirant parti de la ressemblance entre les propriétés des *templates* et celles des langages fonctionnels, on peut, par exemple, envisager un portage C++ efficace d'outils basés sur le modèle BSP, étendant et complétant celui proposé dans [51]. Ce portage fournirait alors une interface *template* adéquate et un mécanisme d'évaluation partielle qui serait capable à la fois de générer le code applicatif final et une fonction numérique capable d'évaluer le temps d'exécution de l'application.

En extrapolant, peut-on montrer, par l'analyse des diverses techniques de programmation *template* couramment utilisées, que les types *templates* sont l'équivalents des fonctions d'ordre supérieur des langages fonctionnels ? Des travaux comme la bibliothèque MPL [2] ou *BOOST:Lambda* [103] montrent que de nombreuses constructions *templates* peuvent être réduites en une série de compositions de telles fonctions et que, réciproquement, des concepts de haut niveau comme le λ -calcul sont exprimables sous forme de *templates* mais peu de travaux théoriques viennent valider cette hypothèse. Nous pensons qu'une formalisation claire de l'isomorphisme d'entre ces deux notions pourraient permettre à ces techniques de programmation d'être utilisées de manière plus systématique et avec des résultats démontrables et ainsi rationaliser leur utilisation. Cette formalisation permettrait alors de rendre accessible un ensemble de modèles et de formalismes issus des langages fonctionnels, et dont l'intérêt théorique ou pratique est avéré, à un large panel de développeurs plus habitué à manipuler des langages impératifs ou orientés objet.

Annexe I

templates et métaprogrammation

Les *templates* – ou modèles – sont un mécanisme du langage C++ dont le but principal est de fournir un support pour la programmation dite «générique», favorisant ainsi la réutilisation de code paramétrable. Un *template* définit une famille de classes ou de fonctions paramétrées par une liste de valeurs ou de types.

I.1 Définitions

I.1.1 Classe *template*

Le listing I.1 présente la définition et l'utilisation d'une classe de tableau *template*. Cette classe de tableau utilise deux paramètres *templates* (ligne 1) : T qui définit le type des éléments stockés dans le tableau et N qui définit la taille du tableau. Lors de l'utilisation de cette classe, l'utilisateur précise la valeur de ces deux paramètres et **instancie** la classe *template* array (lignes 8 et 9).

Plusieurs points sont à noter :

- L'instanciation d'un *template* a lieu lors de la compilation et génère un code temporaire dans lequel les divers paramètres *templates* sont remplacés par leurs valeurs effectives. Ce code intermédiaire est ensuite compilé normalement. Contrairement aux macro-définitions, l'ensemble des vérifications syntaxiques et sémantiques du compilateur sont effectuées.
- Une classe *template* n'existe que lorsque ses paramètres de modèles sont fixés. Ainsi la déclaration :

```
array a;
```

n'a pas de sens est provoqué une erreur de compilation.

- Deux instances du même patron utilisant des paramètres *templates* différents produisent deux types différents. Par exemple, le type `array<double, 3>` est incompatible avec le type `array<int, 8>`.

Listing I.1 – Exemple de classe *template*

```

1  template<typename T, int N> class array
2  {
3      public:
4          static const size_t Size = N;
5
6          array() {}
7          ~array() {}
8          array(const array<T,N>& src);
9          array<T,N> operator=( const array<T,N>& src );
10
11         // ...
12
13         T operator[](size_t i) const { return data[i]; }
14         T& operator[](size_t i) { return data[i]; }
15
16         size_t size() const { return Size; }
17
18     private:
19     T data[N];
20 };
21
22 array<double, 3> t1;
23 array<int, 8>     t2;

```

I.1.2 Fonction *template*

Il est aussi possible de définir des fonctions *templates*. Le listing I.2 définit une fonction calculant la somme d'un tableau. Cette fonction accepte des arguments de n'importe quel type atomique — comme `int` ou `double` — ou d'un type défini par l'utilisateur, à condition que celui-ci fournit une surcharge pour

les opérateurs «==» et «+». On note aussi que l'appel de la fonction à la ligne 9 ne requiert pas de spécification explicite du type du tableau, le compilateur étant capable d'inférer ce dernier en analysant l'appel. Les *template* s'avèrent donc être des outils efficaces pour définir des classes et des fonctions génériques tout en conservant des performances satisfaisantes.

Listing I.2 – Exemple de fonction *template*

```

1 template<int N, typename T> inline T sum( T* array )
2 {
3     T r = 0;
4     for(int i=0; i<N; i++) r += array[i];
5     return r;
6 }
7
8 double a[] = { 1.0, 2.3, 5.0, 6.5 };
9 double s = sum<4>( a );

```

I.1.3 Spécialisation des *templates*

Jusqu'à présent, nous avons défini les classes et les fonctions *templates* d'une manière unique, pour tous les types et toutes les valeurs des paramètres *template*. Il peut néanmoins être intéressant de définir une version particulière d'une classe pour un jeu particulier de paramètres *templates*. Il existe deux types de spécialisation : les spécialisations totales, pour lesquelles il n'y a plus aucun paramètre *template*, et les spécialisations partielles, pour lesquelles seuls quelques paramètres *templates* ont une valeur fixée. Considérons notre exemple de tableau statique : pour diverses raisons, il peut être nécessaire de modifier le contenu de la classe array en fonction du paramètre T. Dans le listing I.3, nous avons spécialisé partiellement la classe array pour gérer les réels double précision via un tableau dynamique.

Lorsque le compilateur tente de résoudre un type *template*, il commence par chercher la spécialisation la plus complète et remonte la liste des spécialisations partielles jusqu'à trouver un cas valide. Il est important de noter que la détermination d'une correspondance entre une signature de fonction ou une déclaration de classe et le *template* correspondant n'utilise que les informations primaires sur les types, c'est à dire qu'il ne peut utiliser les informations concernant les relations d'héritages entre les types pour effectuer un choix de spécialisation.

Listing I.3 – Exemple de spécialisation partielle de *template*

```

1  template<int N> class array<double>
2  {
3      public:
4          static const size_t Size = N;
5
6          array() { data = new double[N]; }
7          ~array() { if(data) delete[] data; }
8          array(const array<double, N>& src);
9          array<double, N> operator=(const array<T, N>& src );
10
11         // ...
12
13         double operator[](size_t i) const { return data[i]; }
14         double& operator[](size_t i) { return data[i]; }
15
16         size_t size() const { return Size; }
17
18     private:
19     T data[N];
20
21 };

```

I.2 Principes de méta-programmation

L'utilisation des classes et fonctions *templates* ne se résument pas à la définition d'outils génériques paramétrables. Il est possible d'exprimer, grâce à diverses constructions classiques, une version statique — c'est à dire résolue à la compilation — de divers éléments dynamiques du langage C++. Parmi ces techniques, on trouve : l'arithmétique statique, les méta-structures de contrôles et les méta-fonctions de manipulation de types.

I.2.1 Arithmétique statique

Les *templates* peuvent être utilisés pour effectuer des calculs sur des entiers ou des valeurs booléennes qui seront évalués lors de la compilation. Le listing I.4 donne un exemple simple d'un tel méta-programme. Ce méta-programme évalue à la compilation la valeur de factorielle N de manière récursive.

Listing I.4 – Factorielle métaprogrammée

```

1 template<int N> struct fac
2 {
3     static const int val = N*fac<N-1>::val;
4 };
5
6 template<> struct fac<0>
7 {
8     static const int val = 1;
9 };

```

Dans une première partie (lignes 1–4), une définition d'un **cas général** est fournie. Cette définition calcule le produit de N avec la valeur de $(N - 1)!$. Une deuxième partie définie **un cas terminal** (lignes 6–9) pour l'évaluation de $0!$ en utilisant une **spécialisation** de la structure initiale. Le listing I.5 montre comment la valeur de $4!$ est calculé par cette structure. A la fin de la compilation, la constante 24 est affectée à la variable `v` et aucun calcul ne sera effectué à l'exécution – ce que confirme l'examen du code assembleur final.

Listing I.5 – Évaluation de `fac<4>::val`

```

1 int v = fac<4>::val;
2 int v = 4 * fac<3>::val;
3 int v = 4 * 3 * fac<2>::val;
4 int v = 4 * 3 * 2 * fac<1>::val;
5 int v = 4 * 3 * 2 * 1 * fac<0>::val;
6 int v = 4 * 3 * 2 * 1 * 1 = 24;

```

L'ensemble des opérations exécutables à la compilation reste limité aux opérations arithmétiques et logiques de bases. Il n'est pas possible par exemple d'appeler des fonctions comme `sqrt` ou `cos` car elles correspondent à du code compilé et utilisable uniquement lors de l'exécution effective du programme.

I.2.2 Structure de contrôle statique

Il est possible de définir des structures de contrôles comme des boucles ou des branchements conditionnels en utilisant des métaprogrammes [186]. Ces métaprogrammes sont basés sur un équivalent récursif des structures originelles en C.

Listing I.6 – Produit scalaire méta-programmé

```

1 template<int N> struct dot
2 {
3     static float product(float* a, float* b)
4     { return a[N-1]*b[N-1] + dot<N-1>::val(a,b); }
5 }
6
7 template<> struct dot<0>
8 {
9     static float product(float* a, float* b)
10    { return a[0]*b[0]; }
11 }
```

Le listing I.6 présente un méta-programme qui évalue le produit scalaire de deux vecteurs de tailles connues en utilisant une forme récursive d'une boucle `for`. La structure définie à la ligne 1 constitue la forme générale de la récursion. Elle évalue la $N^{\text{ième}}$ portion du produit scalaire et l'ajoute à l'évaluation du produit scalaire sur les $N-1$ éléments restants. La structure de la ligne 9 définit l'étape terminale de la récursion en évaluant le produit des derniers éléments du tableau. Ce méta-programme s'utilise comme montré sur le listing I.7

Listing I.7 – utilisation du produit scalaire méta-programmé

```

1 double a[4],b[4];
2 double r = dot<4>::product(a,b);
```

Une fois l'évaluation statique de la structure effectuée, le compilateur génère un code équivalent à celui du listing I.8.

Listing I.8 – Produit scalaire méta-programmé – Code intermédiaire

```
1 double r = a[3]*b[3]+a[2]*b[2]+a[1]*b[1]+a[0]*b[0];
```

I.2.3 Fonctions de manipulations de types

Les *templates* sont aussi capables de manipuler des types comme des variables à part entière. La technique des *Traits* [137] permet de spécifier des méta-programmes qui définissent des «fonctions» qui opèrent sur des types plutôt que

sur des données. Considérons par exemple, le problème du calcul de la moyenne des éléments d'un tableau. Quel type devrait être utilisée pour la valeur de retour ? Si le tableau contient des entiers, le retour devrait être un valeur de type `float`. Mais le type `float` ne convient pas si le tableau contient des complexes ou des doubles par exemple. On résout ce problème en définissant une classe template qui va effectuer une mise en correspondance entre un type d'entrée et le type de retour correspondant (cf listing I.9).

Listing I.9 – Définition d'une classe *Traits*

```

1 template<class T> class avg
2 {
3     typedef T      type_t;
4 }
5
6 template<> class avg<int>
7 {
8     typedef float type_t;
9 }
```

La définition d'un type convenant au calcul de la moyenne d'un tableau d'éléments de type `T` est accessible via `avg<T>::type_t`. La classe définie dans le listing I.9 et sa spécialisation définissent une fonction dont le retour est un type qui dépend du type d'entrée. La fonction générique du calcul de la moyenne des éléments d'un tableau est donné dans le listing I.10. Grâce aux *Traits*, le résultat est correct quelque soit le type utilisé.

Listing I.10 – Application des *Traits* – Moyenne d'un tableau

```

1 template<typename T> typename avg<T>::type_t
2 average( T* tab, int sz )
3 {
4     typename avg<T>::type_t resultat = sum(tab,sz);
5     return resultat/sz;
6 }
```

L'utilisation des *Traits* propose donc une solution générique à l'élaboration de métaprogrammes efficaces car ils permettent de répondre aux exceptions que la simple spécialisation de *templates* ne pourrait résoudre.

I.3 Méta-programmes usuels

Cette section présente plusieurs méta-programmes utilisés de manière courante au sein de l'implantation de E.V.E. et de QUAFF : les adaptateurs valeur/type, les opérateurs logiques statiques, un mécanisme de test de conversion implicite et un mécanisme d'émission d'erreur à la compilation.

I.3.1 Adaptateur valeur/type

De nombreux méta-programmes utilisent des valeurs booléennes ou entières afin de discriminer leur spécialisation. La déclaration de telles spécialisations ne pose pas de problème théorique mais manipuler des valeurs entières prend sensiblement plus de temps que de manipuler des types. Une solution simple consiste à utiliser un adaptateur qui va transformer une valeur numérique constante en type. Cet adaptateur est défini de la manière suivante :

Listing I.11 – boxed : Adaptateur valeur/type

```

1 template<int N> class boxed
2 {
3     static const int value = N;
4 }
5
6 typedef boxed<4> quatre_t;
7 size_t k = quatre_t::value; // k = 4;
```

Très simplement, `boxed` expose un membre statique `value` dont la valeur est fixée à la compilation comme étant égale à la valeur du paramètre `template N`.

I.3.2 Opérateurs logiques statiques

Le choix d'une spécialisation pour une classe `template` complexe est souvent dictée par la combinaison de différents paramètres. Lorsque le nombre de ces paramètres et leurs relations restent faibles, une énumération de l'ensemble des cas possibles suffit à déterminer complètement la liste des spécialisations nécessaires.

Dans les cas où le nombres de paramètres est élevé ou lorsque les relations entre ces derniers définissent un grand nombre de cas identiques, une solution efficace pour minimiser la duplication de code consiste à définir une table de vérité qui énumère l'ensemble des cas possible et d'exprimer pour chacune de ces

spécialisations une équation booléenne qui prend la valeur VRAI lorsque son utilisation est requise. Pour évaluer statiquement ces équations booléennes, il est nécessaire de se pourvoir des éléments permettant leur évaluation. Tout d'abord, il nous faut définir l'équivalent statique des valeurs VRAI et FAUX via la classe boxed.

Listing I.12 – Définition des indicateur de type booléens

```

1 typedef boxed<true>      true_t;
2 typedef boxed<false>    false_t;
```

Ensuite, on se dote de trois opérateurs logiques statique : ET, OU et NON. Leur définition repose sur la spécialisation partielle d'une structure *template* basée sur la table de vérité de chacun de ces opérateurs. Ainsi l'opérateur logique ET renvoie la valeur VRAI si et seulement si ces deux arguments ont pour valeur VRAI. L'opérateur logique OU se définit d'un manière similaire. L'analyse de sa table de vérité nous montre que seul cas FAUX OU FAUX retourne FAUX, tout les autres cas renvoyant VRAI.

Listing I.13 – Opérateurs OU et ET logique statique

```

1 template<class A,class B> struct logical_and
2 {
3     typedef false_t    type_t;
4 };
5
6 template<> struct logical_and<true_t,true_t>
7 {
8     typedef true_t    type_t;
9 };
10
11 template<class A,class B> struct logical_or
12 {
13     typedef true_t    type_t;
14 };
15
16 template<> struct logical_or<false_t,false_t>
17 {
18     typedef false_t    type_t;
19 };
```

Enfin, l'opérateur NON se définit simplement comme :

Listing I.14 – NON logique statique

```

1 template<class A>
2 struct logical_not
3 {
4     typedef true_t type_t;
5 };
6
7 template<>
8 struct logical_not<true_t>
9 {
10    typedef false_t type_t;
11 };

```

D'autres opérateurs de ce type — comme XOR, NAND ou NOR — sont définissables en utilisant une technique similaire ou en combinant les éléments déjà définis et permettent de compléter ce panel de méta-fonctions.

I.3.3 Test de conversion implicite

Considérons une hiérarchie de classe quelconque et une classe *template* nécessitant d'être spécialisé pour n'importe quel type appartenant à cette hiérarchie.

```

1 struct A {};
2 struct B: public A {};
3
4 template<class T> struct Foo
5 {
6     void bar() { cout << "bar" << endl; }
7 };

```

Lors de la définition de type ou classes *template*, le mécanisme de résolution des spécialisations *template* ne peut, de par sa nature statique, détecter ces relations d'héritages. Ainsi, la stratégie naïve de surcharge ne fonctionne pas. Il est donc nécessaire de déterminer un mécanisme statique permettant de détecter qu'un type donné appartient à une hiérarchie de classe afin d'éviter de multiplier les spécialisations partielles pour chaque classe de la hiérarchie.

```

1 template<> struct Foo<A>
2 {
3     void bar() { cout << "A bar" << endl; }
4 }
```

Ce mécanisme repose sur un constat simple. Si l'on considère un pointeur vers une classe A — pA — et un pointeur vers une classe B — pB — , le fait de pouvoir affecter pB à pA suffit à démontrer que A et B appartiennent à une même hiérarchie de classe. Comment transposer cette relation au sein d'une évaluation statique ? Le standard du C++ travaille pour nous en fournissant l'opérateur `sizeof`. `sizeof` est un opérateur dont la puissance est souvent mal estimée. `sizeof` est en effet défini afin de retourner la taille en octets de son opérande. Un fait peu connu est que `sizeof` effectue ce calcul de manière statique et ce **quelque soit le type et la complexité de son opérande**. Grâce à cette propriété, nous définissons la métा-fonction `conversion`. Cette fonction permet de déterminer statiquement si il existe une conversion implicite entre les deux types T et U. Pour ce faire, elle utilise un certains nombre de types et de fonctions qui vont permettre de mener à bien ce test.

Listing I.15 – Méta-detecteur de conversion

```

1 template<class T, class U> struct conversion
2 {
3     typedef char Small;
4     struct Big { char dummy[2]; };
5
6     static Big Test(...);
7     static Small Test(U);
8     static T MakeT();
9
10    static const size_t size_U = sizeof(Small);
11    static const size_t size_T = sizeof(Test(MakeT()));
12    static const bool exists = (size_U == size_T);
13    typedef boxed<exists> type_t;
14}
```

Le résultat de cette métaprogramme est contenu dans la valeur booléenne statique `exists` qui évalue l'égalité entre les quantités `size_U` et `size_T`. `size_U` est égale à la taille en octet du type `Small`, c'est à dire 1. `size_T` est plus com-

plexé et demande l'évaluation de la taille en octet du type renvoyé par l'expression `Test (MakeT ())`. `MakeT ()` renvoyant systématiquement une instance `T`, deux cas se présentent alors :

- Si il existe une conversion entre `T` et `U`, alors la surcharge de `Test` qui prend en argument un `U` va être sélectionnée et renvoyer un élément de type `Small`. La valeur de `size_T` devient alors 1 et `exists` prend la valeur `true`.
- Si `T` ne peut être convertie en `U`, alors la surcharge de `Test` utilisant une liste indéfinie d'argument est sélectionnée car elle correspond selon le standard au plus mauvais cas de conversion possible. `Test` renvoie alors un élément de type `Big`, la valeur de `size_T` devient alors 2 et `exists` prend la valeur `false`.

Ce qui rend cette approche possible est que l'ensemble de ces évaluations sont effectuées par `sizeof` sans jamais avoir besoin d'exécuter le code d'aucune fonction. `sizeof` effectue en effet une analyse poussée des types qui lui sont passé en argument et utilise la totalité des sous-systèmes du compilateur (analyseur syntaxique, convertisseur de type) afin de mener son calcul à bien.

Comment alors détecter si une classe appartient à une hiérarchie ? Il suffit d'utiliser la méta-fonction `conversion` comme illustré dans le listing I.16

Listing I.16 – Méta-detecteur de conversion

```

1  struct A           {};
2  struct B: public A {};
3
4  template<class T, class ST = conversion<T*,A*>::type_t
5  struct Foo
6  {
7      void bar() { cout << "bar" << endl; }
8  };
9
10 template<class T> struct Foo<T,true_t>
11 {
12     void bar() { cout << "A bar" << endl; }
13 };

```

Le paramètre `template class ST = conversion<T*,A*>::type_t` teste sta-

tiquement si une conversion entre les types T^* et A^* existe, ce qui est équivalent à tester la possibilité d'une affectation entre des pointeurs de ce type. Dans le cas général, la classe `Foo` conserve son comportement classique. Si `ST` prend la valeur `true_t`, alors sa spécialisation pour les classes héritant de `A` est sélectionnée.

I.3.4 Levée d'erreurs à la compilation

Lors de différents tests effectués au sein de méta-fonctions, il est parfois souhaitable de stopper la compilation au lieu de produire un code exécutable dont le comportement est incorrect. Une manière simple de définir de tels messages passe par l'utilisation de la spécialisation partielle des *templates*. Considérons la classe suivante :

Listing I.17 – Classe *template* incomplète

```
1 template<bool V> struct static_assert;
2 template<> struct static_assert<true> { };
```

Si l'on tente d'instancier cette classe avec la valeur `true`, tout se passe bien. Si l'on tente d'instancier cette classe avec la valeur `false`, le compilateur est dans l'impossibilité de générer le code nécessaire et produit une erreur. Reste alors à pouvoir spécifier le contenu de l'erreur. La macro `STATIC_ASSERT` met en oeuvre ce système.

Listing I.18 – Générateur d'erreur à la compilation

```
1 #define STATIC_ASSERT(M, C) { static_assert<C> M; }
```

Comment cette macro fonctionne-t-elle ? Ces deux arguments correspondent respectivement à un identifiant d'instance et à une condition qui s'évalue comme un booléen. Lors de l'appel à cette macro, le compilateur tente d'instancier la classe `static_assert` en lui affectant la valeur de `C`. Si cette valeur est évaluée à `false`, le message d'erreur suivant est émis par le compilateur. Ainsi, l'appel :

```
STATIC_ASSERT(INVALID_TYPE_SIZE, (sizeof(*p) == 4));
```

produit le message suivant :

```
Error : INVALID_TYPE_SIZE -- Cannot instantiate incomplete type
```

à la ligne précise ou la macro STATIC_ASSERT a été utilisée si la taille en octet de `*p` n'est pas égale à 4.

La question de son impact sur le code final est réglée élégamment. Si `COND` s'évalue à `true`, une instance inutilisé de `static_assert` est créée. Or la plupart des compilateurs vont optimiser cette création inutile et la retirer entièrement du code exécutable final.

I.4 Discussion

Les différentes techniques de méta-programmations que nous avons présentées nous permettent de résoudre une grande partie des problèmes de performances soulevés dans ce manuscrit. Ces différentes techniques nous montrent aussi que :

- Il existe une asymétrie très forte entre les deux niveaux de lecture du code *template C++*. En effet, il n'existe aucun moyen de manipuler des nombres réels, des objets ou des entrées-sorties de types fichiers.
- Il n'est pas possible de contraindre les instanciations *template* à n'accepter qu'un sous-ensemble de type comme paramètres. Cette limitation laisse l'utilisateur final seul face à des messages d'erreur de compilations relativement complexes et peu lisibles.
- La possibilité de manipuler des types comme des variables et de pouvoir générer des fonctions opérant sur ces derniers est un des points forts des diverses techniques de métaprogrammation. Des techniques comme les *Traits* ou les *Expression Templates* sont des exemples pertinents de la puissance de cette fonctionnalité.

Annexe II

prime_sieve par E. Unruh

Le principe de ce métaprogramme écrit par Erwin Unruh [181] est d'afficher lors de la compilation une liste de nombres premiers. A chaque appel de la fonction `Prime_print::f()`, le compilateur tente d'initialiser la variable `d`. Cette initialisation échoue tant que la valeur passé au constructeur de `d` n'est pas égale à 0, qui est la seule valeur entière pouvant être transtypée en `void*`. Or, cela ne se produit que lorsque la constante `prim` est non nulle, c'est à dire si le nombre passé en argument à `Val <>` est non premier. Le résultat de la compilation du listing II.1 est donné ci-dessous.

```
> g++ prime.cpp -o prime
error line 18 :
prime.cpp: conversion from int to non-scalar type Val<17> requested
prime.cpp: conversion from int to non-scalar type Val<13> requested
prime.cpp: conversion from int to non-scalar type Val<11> requested
prime.cpp: conversion from int to non-scalar type Val<7> requested
prime.cpp: conversion from int to non-scalar type Val<5> requested
prime.cpp: conversion from int to non-scalar type Val<3> requested
prime.cpp: conversion from int to non-scalar type Val<2> requested
```

Listing II.1 – Calculs des N premiers nombres premiers

```

1  template<int p, int i> struct is_prime
2  {
3      enum {      prim = (p==2) || (p%i)
4          && is_prime<(i>2?p:0),i-1>::prim
5      };
6  };
7
8  template<> struct is_prime<0,0> { enum {prim=1}; };
9  template<> struct is_prime<0,1> { enum {prim=1}; };
10
11 template<int i> struct Val { Val(void*); };
12
13 template<int i> struct Prime_print
14 {
15     Prime_print<i-1> a;
16     enum { prim = is_prime<i,i-1>::prim };
17
18     void f()
19     {
20         Val <i> d = prim ? 1 : 0;
21         a.f();
22     }
23 };
24
25 template<> struct Prime_print<1>
26 {
27     enum { prim=0 };
28     void f()
29     {
30         Val <1> d = prim ? 1 : 0;
31     }
32 };
33
34 int main()
35 {
36     Prime_print<18> a;
37     a.f();
38     return 0;
39 }
```

Annexe III

Exemple de fonction ALTIVEC complexe

Les extraits de code présentés ici définissent la version scalaire et vectorielle d'une fonction appliquant un filtre gaussien 1x3 à un signal monodimensionnel. L'algorithme AltiVec est basé sur un schéma de chargement ad hoc qui permet de filtrer le signal source par bloc de huit valeurs avec seulement deux chargements et trois opérations de type multiplication-addition. Le lecteur appréciera la différence notable de complexité entre le code C classique et le code C présenté dans le listing III.1 et celui utilisant des primitives AltiVec présenté dans le listing III.2.

Listing III.1 – Application d'un filtre gaussien 1X3 – Code C

```
1 void C_filter( char* in, short* out, int SIZE)
2 {
3     int bord = SIZE-1;
4
5     // Gestion des bords
6     out[0]      = (2*in[0]+in[1])/4;
7     out[bord]   = (in[bord-1]+2*in[bord])/4;
8
9     // Boucle principale
10    for(size_t i=1; i<bord; i++)
11    {
12        out[i] = (in[i-1]+2*in[i]+in[i+1])/4;
13    }
14 }
```

Listing III.2 – Application d'un filtre gaussien 1X3 – Code AltiVec

```

1  void AV_filter( char* img, short* res, int SIZE)
2  {
3      vector unsigned char zu8,t1,t2,t3,t4;
4      vector signed short zs16,x1h,x1l,x2h,x2l;
5      vector signed short x3h,x3l,rh,rl,v0,v1,shift;
6
7      v0      = vec_splat_s16(2);
8      v1      = vec_splat_s16(4);
9      shift   = vec_splat_s16(8);
10     zu8     = vec_splat_u8(0);
11     zs16    = vec_splat_s16(0);
12
13     for( int j = 0; j< SIZE/16 ; j++ )
14     {
15         t1 = vec_ld(j*16, img);
16         t2 = vec_ld((j+1)*16, img);
17         t3 = vec_sld(t1,t2,1);
18         t4 = vec_sld(t1,t2,2);
19
20         x1h = vec_mergesh(zu8,t1);
21         x1l = vec_mergel(zu8,t1);
22         x2h = vec_mergesh(zu8,t3);
23         x2l = vec_mergel(zu8,t3);
24         x3h = vec_mergesh(zu8,t4);
25         x3l = vec_mergel(zu8,t4);
26
27         rh = vec_mladd(x1h,v0,zs16);
28         rl = vec_mladd(x1l,v0,zs16);
29         rh = vec_mladd(x2h,v1,rh);
30         rl = vec_mladd(x2l,v1,rl);
31         rh = vec_mladd(x3h,v0,rh);
32         rl = vec_mladd(x3l,v0,rl);
33
34         rh = vec_sr(rh,shift);
35         rl = vec_sr(rl,shift);
36         t1 = vec_packsu(rh,rl);
37         vec_st(t1,j,out);
38     }
39 }
```

Annexe IV

Opérations sur les listes statiques de types

Nous présentons ici le code complet des méta-programmes de manipulations des classes `typelist` et `tuple` présentées dans la section 5.3.1.2. Toutes sont basées sur l'exploitation du caractère récursif de la définition de `typelist` et des techniques de spécialisations partielles des classes *templates*.

Les fonctions présentées sont :

- l'évaluation de la taille d'une liste de type;
- l'accès aléatoire aux éléments d'une liste de type;
- le retrait d'élément à une liste de type;
- l'ajout d'élément à une liste de type;
- l'extraction d'une sous-liste;
- l'application d'une méta-fonction sur les éléments d'une liste de type;
- l'accès aléatoire aux éléments d'un tuple.

```

1  template<class TL>
2  struct length;
3
4  template<class H, class T>
5  struct length< typelist<H,T> >
6  {
7      static const size_t value = 1 + length <T>::value;
8  };
9
10 template<>
11 struct length< null_t >
12 {
13     static const size_t value = 0;
14 };

```

IV.2 Accès aléatoire aux éléments d'une liste de type

```

1  template<class TL, size_t I>
2  struct type_at;
3
4  template<class H, class T>
5  struct type_at<typelist<H,T>, 0>
6  {
7      typedef H type_t;
8  };
9
10 template<class H, class T, size_t I>
11 struct type_at<typelist<H,T>, I>
12 {
13     typedef typename type_at<T,I-1>::type_t type_t;
14 };
15
16 template<size_t I> struct type_at<null_t,I>
17 {
18     return null_t;
19 };

```

IV.3 Application d'une méta-fonction à une liste

```

1  template<class TL, template<class> class F>
2  struct type_map;
3
4  template<class H, class T, template<class> class F>
5  struct type_map<typelist<H, T>, F>
6  {
7      typedef typename F<H>::type_t head_t;
8      typedef typename type_map<T, F>::type_t tail_t;
9      typedef typename append<head_t, tail_t>::type_t type_t;
10 }
11
12 template<class TL, template<class> class F>
13 struct type_map<null_t, F>
14 {
15     typedef null_t type_t;
16 }
```

IV.4 Retrait d'élément à une liste de type

```

1  template<class TL, class V> struct erase;
2
3  template<class T, class V> struct erase<typelist<V, T>, V>
4  {
5      typedef T type_t;
6  };
7
8  template<class H, class T, class V>
9  struct erase<typelist<H, T>, V>
10 {
11     typedef typelist<H, typename erase<T, V>::type_t>
12         type_t;
13 };
14
15 template<class V> struct erase<null_t, V>;
16 {
17     typedef null_t type_t;
18 }
```

```
1  template<class TL, class V>
2  struct append;
3
4  template<class V>
5  struct append<null_t,V>
6  {
7      typedef typelist<V,null_t> type_t;
8  };
9
10 template <class H, class T>
11 struct append<null_t, typelist<H, T> >
12 {
13     typedef typelist<H, T> type_t;
14 };
15
16 template <class H, class T, class V>
17 struct append<typelist<H,T>,V>
18 {
19     typedef typelist<H,typename append<T, V>::type_t>
20         type_t;
21 };
```

IV.6 Extraction d'une sous-liste

```
1 template<class TL, size_t B, size_t E>
2 struct extract;
3
4 template<class H, class T, size_t B, size_t E>
5 struct extract< typelist<H,T>,B,E >
6 {
7     typedef typelist<H,T> in_t;
8     typedef typename type_at<in_t,B>::type_t el_t;
9     typedef typename extract<in_t,B+1,E>::type_t ex_t;
10    typedef typelist<el_t,ext_t> type_t;
11 };
12
13 template<class H, class T, size_t E>
14 struct extract< typelist<H,T>,E,E>
15 {
16     typedef typelist<H,T> in_t;
17     typedef typename type_at<in,E>::type_t el_t;
18     typedef typelist<el_t, null_t> type_t;
19 };
20
21 template<size_t B, size_t E>
22 struct extract<null_t,B,E>
23 {
24     typedef null_t type_t;
25 };
```

```

1  template <class TP, size_t I>
2  struct finder
3  {
4      typedef typename TP::list_t list_t;
5      typedef typename type_at<list_t, I>::type_t base_t;
6      typedef typename reference_to<base_t>::type_t type_t;
7      typedef typename TP::tail_t tail_t;
8
9      static type_t Do(TP& obj)
10     {
11         return finder<tail_t, I-1>::Do((tail_t&) (obj));
12     }
13 }
14
15 template<class TP>
16 struct finder<TP, 0>
17 {
18     typedef typename TP::list_t list_t;
19     typedef typename list_t::head_t base_t;
20     typedef typename reference_to<base_t>::type_t type_t;
21
22     static type_t Do(TP& obj) { return obj.mHead; }
23 }
24
25 template <size_t I, class H>
26 typename finder<H, I>::type_t field(H& obj)
27 {
28     return finder<H, I>::Do(obj);
29 }
```

Bibliographie

- [1] F. Ababsa, D. Roussel, M. Mallem, and J.-Y. Didier. 3d free form objects recognition technique using photoclinometry. In *Proceedings of the 4th International Workshop on Advanced Concepts for Intelligent Vision Systems (ACIVS 2002)*, pages 130–135, 9-11 sep 2002.
- [2] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.
- [3] ABSOFT. VAST Code Optimizer - Site Commercial de l'éditeur.
<http://www.absoft.com/Products/Libraries/vast.html>.
- [4] M. Aldinucci. The meta Transformation Tool for Skeleton-Based languages. In S. Gorlatch and C. Lengauer, editors, *Proc. of 2nd Intl. Workshop on Constructive Methods for Parallel Programming (CMPP2000)*, pages 53–68. Fakultät für mathematik und informatik, Uni. Passau, Germany, July 2000.
- [5] M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In S. Gorlatch, editor, *CMPP'98: First International Workshop on Constructive Methods for Parallel Programming*, 1998.
- [6] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems*, MIT, Boston, USA, 1999. IASTED/ACTA press.
- [7] M. Aldinucci and M. Danelutto. An operational semantics for skeletons. Technical Report TR-02-13, Computer Science Department, University of Pisa, Italy, July 2002.
- [8] Christopher Alexander. *A Pattern Language*. Oxford University Press, 1977.

- [9] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.
 - [10] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. AW C++ in Depth Series. Addison Wesley, January 2001.
 - [11] Vassil N. Alexandrov and Jack Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*. 5th European PVM/MPI Users' Group Meeting, Springer, September 1998.
 - [12] J. Allard, J.-S. Franco, C. Ménier, and B. Raffin. Marker-less real time 3d modeling for virtual reality. In *Immersive Projection Technology*, 2004.
 - [13] J. Allard, J.-S. Franco, C. Ménier, and B. Raffin. The grimage platform: A mixed reality environment for interactions. In *International Conference on Vision Systems*, 2006.
 - [14] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conf. Proc.*, 30:483–485, 1967.
 - [15] T. Anderson, D. E. Culler, and D. A. Patterson. A case for networks of workstations:now. In *IEEE Micro*, 1995.
 - [16] R. Andraka. FIR filter fits in an FPGA using a bit-serial approach, 1993.
 - [17] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, 1983.
 - [18] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing '90*, pages 2–11, New York, November 1990.
 - [19] Remzi H. Arpacı, David E. Culler, Arvind Krishnamurthy, Steve G. Steinberg, and Katherine Yelick. Empirical evaluation of the cray-t3d: a compiler perspective. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 320–331. ACM Press, 1995.
 - [20] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for on-line non-linear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, February 2002.
 - [21] Nicholas Ayache and Francis Lustman. Trinocular stereo vision for robotics. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(1):73–85, 1991.
 - [22] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3l: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7:225–255, 1995.
-

- [23] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language algol 60. *Commun. ACM*, 3(5):299–314, 1960.
 - [24] Dana Ballard and Chris Brown. *Computer vision*. Prentice Hall eds., 1982.
 - [25] P. Beardsley, P. Torr, and A. Zisserman. 3d model acquisition from extended image sequences. In *European Conference on Computer Vision*, pages 683–695, 1996.
 - [26] Kent Beck and Ralph E. Johnson. Patterns generate architectures. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 139–149, London, UK, 1994. Springer-Verlag.
 - [27] Anne Benoit and Murray Cole. Two fundamental concepts in skeletal parallel programming. In V. Sunderam, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2005), Part II*, LNCS 3515, pages 764–771. Springer Verlag, 2005.
 - [28] G. Blanc, T. Chateau, Y. Mezouar, O. Ait-Ader, P. Martinet, L. Eck, V. Moreau, and A. Nadim. Implementation of a vision based navigation framework on a house mobile robot prototype. In *International Conference on Advances in Intelligent Systems - Theory and Applications, AISTA'04, ISBN 2-9599776-8-8*, Kirchberg, Luxembourg, 15-18 Novembre 2004.
 - [29] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iwarpc: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.
 - [30] Zeki Bozkus, Geoffrey Fox, and Sanjay Ranka. Modelling the cm-5 multicomputer. In *Proceedings of Frontiers '92*, 1992.
 - [31] Silvia Breitinger, Ulrike Klusik, and Rita Loogen. From (sequential) haskell to (parallel) eden: An implementation point of view. In *Principles of Declarative Programming, 10th International Symposium, PLILP'98 Held Jointly with the 7th International Conference, ALP'98*, pages 318–334, 1998.
 - [32] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.
-

- [33] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, 1999.
- [34] C. Papageorgiou, M. Oren, and T. Poggio. A general framework for object detection. In *IEEE Conference on Computer Vision*, pages 555–562, 1998.
- [35] Franck Cappello and Daniel Etiemble. Mpi ou mpi+openmp sur grappes de multiprocesseurs? *Technique et Science Informatiques*, 21(2):253–272, 2002.
- [36] Franck Cappello, Olivier Richard, and Daniel Etiemble. Investigating the performance of two programming models for clusters of smp pcs. In *HPCA*, pages 349–359, 2000.
- [37] T. Chateau, V. Gay-Belille, F. Chausse, and J.T. Lapresté. Real-time tracking with classifiers. In *WDV Workshop on Dynamical Vision at ECCV2006*, 2006.
- [38] Shigeru Chiba. A metaobject protocol for c++. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, New York, NY, USA, 1995. ACM Press.
- [39] E. Chow and D. Hysom. Assessing performance of hybrid MPI/openMP programs on SMP clusters. Technical Report UCRL-JC-143957, Lawrence Livermore National Laboratory, May 2001. http://www.llnl.gov/CASC/people/chow/chow_pubs.html.
- [40] D. Chung, R. Hirata, T. N. Mundhenk, J. Ng, R. J. Peters, E. Pichon, A. Tsui, T. Ventrice, D. Walther, P. Williams, and L. Itti. A new robotics platform for neuromorphic vision: Beobots. In *Lecture Notes in Computer Science*, volume 2525, pages 558–566, Nov 2002.
- [41] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. Anacleto: A template-based p3l compiler. In *Proceedings of the PCW'97*, 1997.
- [42] David Clark. Openmp: A parallel standard for the masses. *IEEE Concurrency*, 6(1):10–12, 1998.
- [43] M. Cole. *Research Directions in Parallel Functional Programming*, chapter 13, Algorithmic skeletons. Springer, 1999.
- [44] M. Cole. Bringing skeletons out of the closet : A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 3:389–406, 2004.
- [45] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1989.

- [46] Apple Developper Connection. Apple velocity engine – vector libraries. http://developer.apple.com/hardwaredrivers/ve/vector_libraries.html.
 - [47] Rémi Coudarcher. *Composition de squelettes algorithmiques : application au prototypage rapide d'applications de vision*. PhD thesis, Université Blaise Pascal, Clermont-Ferrand, 2002.
 - [48] J.D. Crisman and J.A. Webb. The warp machine on navlab. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13:451–465, 1991.
 - [49] Danny Crookes. Architectures for high performance image procesing: the future. *Journal of Systems Architecture*, 45(10):739–748, 1999.
 - [50] Krzysztof Czarnecki. Generative programming: Methods, techniques, and applications. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 351–352, London, UK, 2002. Springer-Verlag.
 - [51] F. Dabrowski and F. Loulergue. Functional bulk synchronous programming in c++. In *21st IASTED International Multi-conference, Applied Informatics (AI 2003), Symposium on Parallel and Distributed Computing and Networks*, pages 462–467. ACTA Press, february 2003.
 - [52] L. Damez. Evaluation de la Technologie AltiVec pour les Algorithmes de Vision Bas Niveau. Master's thesis, Université Blaise Pascal, Clermont-Ferrand, Septembre 2002.
 - [53] M. Danelutto. On skeletons and design patterns. In *PARCO 2001. Special Session on Parallel and Distributed Image and Video Processing (ParIm'01)*, Naples, Italie, Septembre 2001.
 - [54] Marco Danelutto and Patrizio Dazzi. Joint structured/unstructured parallelism exploitation in muskel. In *Proceedings of ICCS 2006 / PAPP 2006*, 2006.
 - [55] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In *Parallel Architectures and Languages Europe*. Springer-Verlag, 1993.
 - [56] Joel de Guzman. The SPIRIT Parser Framework –. <http://www.boost.org/libs/spirit/index.html>, 2003.
 - [57] Jean Pierre Dérutin, Fabio Dias, Lionel Damez, and Nicolas Allezard. Simd, smp and mimd-dm parallel approaches for real-time 2d image stabilization. In *CAMP*, pages 73–80, 2005.
 - [58] Keith Diefendorf. AltiVec Extension to Power PC Accelerates Media Processing. Technical report, IEEE Micro, 2001.
-

- [59] Bruce A. Draper, J. Ross Beveridge, A. P. Willem Bohm, Charles Ross, and Monica Chawathe. Implementing image applications on FPGAs. In *ICPR '02: Proceedings of the 16 th International Conference on Pattern Recognition (ICPR'02) Volume 3*, page 30265. IEEE Computer Society, 2002.
 - [60] T. Drayer. A design methodology for creating programmable logic-based real-time image processing hardware, 1997.
 - [61] Z. Duric and A. Rosenfeld. Shooting a smooth video with a shaky camera. *Machine Vision and Applications*, 13:303–313, 2003.
 - [62] J.-D. Durou. L'équation eikonale. In *Actes des 4èmes Journées ORASIS*, pages 34–39, Mulhouse, France, October 1993. (in french).
 - [63] Dawson R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of USENIX Conference on Domain-Specific Languages*, pages 103–118, 1997.
 - [64] G. E. Fagg, J. J. Dongarra, and A. Geist. Heterogeneous MPI application interoperation and process management under PVMPI. *Lecture Notes in Computer Science*, 1332:91–98, 1997.
 - [65] Graham Fagg and Jack Dongarra. PVMPI: An integration of PVM and MPI systems. *Calculateurs Parallèles*, 8(2):151–166, 1996.
 - [66] Joel Falcou. C+FOX – high performance open source firewire camera driver. <http://cfox.sourceforge.net>, 2003.
 - [67] Joel Falcou, Thierry Chateau, Jocelyn Sérot, and J.T. Lapresté. QUAFF: Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(7-8):604–615, Septembre 2006.
 - [68] Joel Falcou, Thierry Chateau, Jocelyn Sérot, and J.T. Lapresté. Real time parallel implementation of a particle filter based visual tracking. In *CIMCV 2006 - Workshop on Computation Intensive Methods for Computer Vision at ECCV 2006*, Grazz, 2006.
 - [69] Joel Falcou and Jocelyn Sérot. Camlg4: une bibliothèque de calcul de parallèle pour objective caml. In *Journées Francophones des Langages Applicatifs*, 2003.
 - [70] Joel Falcou and Jocelyn Sérot. E.V.E., an object oriented simd library. In *Proceedings of Practical Aspects of High-level Parallel Programming, ICCS 2004*, volume 3, pages 323–330, 2004.
 - [71] Joel Falcou and Jocelyn Sérot. E.V.E., An Object Oriented SIMD Library. *Scalable Computing: Practice and Experience*, 6(4):31–41, December 2005.
-

- [72] Joel Falcou, Jocelyn Sérot, Thierry Chateau, and Frédéric Jurie. Un cluster de calcul hybride pour les applications de vision temps réel. In *Proceedings of GRETSI'05*, Louvain-la-Neuve, September 2005.
 - [73] Joel Falcou, Jocelyn Sérot, Thierry Chateau, and Frédéric Jurie. A parallel implementation of a 3d reconstruction algorithm for real-time vision. In *PARCO 2005 - ParCo, Parallel Computing*, Malaga, Spain, September 2005.
 - [74] Abdelkrim Fatni, Dominique Houzet, and Jean-Luc Basille. The c// data parallel language on a shared memory multiprocessor. In IEEE Computer Society Press, editor, *CAMP'97*, pages 10–15, Los Alamitos, 1997. Octobre.
 - [75] Martin A. Fischler and Oscar Firschein. *Intelligence: the eye, the brain, and the computer*. Addison-Wesley Longman Publishing Co., Inc., 1987.
 - [76] B. G. Fitch, R. S. Germain, M. Mendell, J. Pitera, and M. Pitman. Blue matter, an application framework for molecular simulation on blue gene. *Journal of Parallel and Distributed Computing*, 63:759–773, 2003.
 - [77] Mickael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computing*, 21(9):948–963, 1972.
 - [78] High Performance Fortran Forum. High performance fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.
 - [79] MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Application*, 8:165–416, 1994.
 - [80] J.-S. Franco, C. Ménier, E. Boyer, and B. Raffin. A distributed approach for real-time 3d modeling. In *IEEE CVPR Workshop on Real-Time 3D Sensors and their Applications*, 2004.
 - [81] Jean-Sébastien Franco and Edmond Boyer. Une approche hybride pour calculer l’enveloppe visuelle d’objets complexes. In *Actes des Journées ORASIS, Gerardmer*, pages 67–74, May 2003.
 - [82] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *European Conference on Computational Learning Theory*, pages 23–37, 1995.
 - [83] Andrea Fusiello, Emanuele Trucco, and Alessandro Verri. A compact algorithm for rectification of stereo pairs. *Machine Vision and Applications*, 12(1):16–22, 2000.
-

- [84] Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
 - [85] A. Geist, A. Beguelin, Jack Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994.
 - [86] Robert S. Germain, Blake Fitch, Aleksandr Rayshubskiy, Maria Eleftheriou, Michael C. Pitman, Frank Suits, Mark Giampapa, and T.J. Christopher Ward. Blue matter on blue gene/l: massively parallel computation for biomolecular simulation. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 207–212, New York, NY, USA, 2005. ACM Press.
 - [87] Dominique Ginhac. *Prototypage rapide d'applications paralleles de vision artificielle par squelettes fonctionnels*. PhD thesis, Université Blaise Pascal, Clermont-Ferrand, 1999.
 - [88] Dominique Ginhac, Jocelyn Sérot, and Jean-Pierre Dérutin. Fast prototyping of image processing applications using functional skeletons on mimd-dm architecture. In *IAPR Workshop on Machine Vision Applications*, pages 468–471, 1998.
 - [89] Douglas Gregor and al. The boost c++ library. <http://boost.org/>, 2003.
 - [90] J.L. Gustafson. Reevaluating amdahl's law. *CACM*, 31:532–533, 1988.
 - [91] Yoshimoto H., D. Arita, and R. Taniguchi. Real-time image processing on ieee1394-based pc cluster. In *15th International Parallel and Distributed Processing Symposium*, 2001.
 - [92] M. Hamdan, G. Michaelson, and P. King. A scheme for nesting algorithmic skeletons. In *Proceedings of the 10th International Workshop on Implementation of Functional Languages, IFL '98*, pages 195–212, 1998.
 - [93] Scott W. Haney. Is c++ fast enough for scientific computing? *Comput. Phys.*, 8(6):690–694, 1994.
 - [94] C. Harris and M. Stephens. A combined corner and edge detector. In *4th ALVEY Vision Conference*, pages 147–151, 1988.
 - [95] R. Hartley and R. Gupta. Computing matched-epipolar projections. In *CVPR93*, pages 549–555, 1993.
 - [96] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521623049, 2000.
-

- [97] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.
 - [98] W. Daniel Hillis. *The connection machine*. MIT Press, Cambridge, MA, USA, 1986.
 - [99] Jürg Hutter and Alessandro Curioni. Dual-level parallelism for ab initio molecular dynamics: Reaching teraflop performance with the cpmd code. *Parallel Computing*, 1:1–17, 2005.
 - [100] Michael Isard and Andrew Blake. Condensation – conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29(1):5–28, 1998.
 - [101] Michael Isard and John MacCormick. Bramble: A bayesian Multiple-Blob tracker. In *Proceedings of International Conference on Computer Vision*, volume 2, pages 34–41, 2001.
 - [102] Yutaka Ishikawa, Atsushi Hori, Mitsuhsisa Sato, and Motohiko Matsuda. Design and implementation of metalevel architecture in c++ - mpc++ approach. In *Proceedings of the Reflection '96 Conference.*, pages 153–166, San Francisco, USA, 1996.
 - [103] Jaakko Järvi, Gary Powell, and Andrew Lumsdaine. The lambda library: unnamed functions in c++. *Softw. Pract. Exper.*, 33(3):259–291, 2003.
 - [104] Jesse S. Jin, Zhigang Zhu, and Guangyou Xu. Digital video sequence stabilization based on 2.5d motion estimation and inertial motion filtering. *Real-Time Imaging*, 7(4):357–365, 2001.
 - [105] Eric E. Johnson. Completing an MIMD multiprocessor taxonomy. *SIGARCH Computer Architecture News*, 16(3):44–47, 1988.
 - [106] Neil D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.
 - [107] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
 - [108] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME-Journal of Basic Engineering*, 82(Series D):35–45, 1960.
 - [109] T. Kanade, P. J. Narayanan, and P. W. Rander. Virtualized reality: concepts and early results. In *VSR '95: Proceedings of the IEEE Workshop on Re-*
-

- presentation of Visual Scenes*, page 69, Washington, DC, USA, 1995. IEEE Computer Society.
- [110] Takeo Kanade, Hideo Saito, and Sundar Vedula. The 3d room: Digitizing time-varying 3d events by synchronized multiple video streams. Technical Report CMU-RI-TR-98-34, Robotics Institute, Carnegie Mellon University, December 1998.
 - [111] S Karmesin and al. Arrays Design and Expression Evaluation in POOMA II. *ISCOPE'98*, 1505:38–44, 1998.
 - [112] Robert C. Kirby. A new look at expression templates for matrix computation. *Computing in Science and Engineering*, 05(3):66–70, 2003.
 - [113] R. L. Kirk, J. M. Barrett, and L. A. Soderblom. Photoclinometry made simple...? In *Proceedings of the ISPRS Commission IV Symposium*, volume XXXIV-4 of *International Archives of Photogrammetry and Remote Sensing*, Houston, Texas, USA, March 2003.
 - [114] D. C. Knill and D. Kersten. Learning a near-optimal estimator for surface shape from shading. *Computer Vision, Graphics, and Image Processing*, 50(1):75–100, April 1990.
 - [115] C. Koebel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, 1994.
 - [116] Géraud Krawezik, Guillaume Alléon, and Franck Cappello. Spmd openmp versus mpi on a ibm smp for 3 kernels of the nas benchmarks. In *ISHPC*, pages 425–436, 2002.
 - [117] Argy Krikellis and Mitsuhsisa Sato. Compas: A pc-based smp cluster. *IEEE Concurrency*, 7(1):82–86, 1999.
 - [118] H. Kuchen and J. Striegnitz. Higher-order functions and partial applications for a C++ skeleton library. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 122–130, 2002.
 - [119] Herbert Kuchen. A skeleton library. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, London, UK, 2002. Springer-Verlag.
 - [120] L. Lacassagne, D. Etiemble, and S.A. Ould Kablia. 16-bit floating point instructions for embedded multimedia applications. *camp*, 00:198–203, 2005.
 - [121] Jean-Marc Lavest, Marc Viala, and Michel Dhome. Do we really need an accurate calibration pattern to achieve a reliable camera calibration? In *ECCV '98: Proceedings of the 5th European Conference on Computer Vision-Volume I*, pages 158–174, London, UK, 1998. Springer-Verlag.

- [122] M. Lhuillier and L. Quan. A quasi-dense approach to surface reconstruction from uncalibrated images. *Transactions on Pattern Analysis and Machine Intelligence*, 27(3):418–433, 2005.
- [123] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Peter Wyckoff, and D K. Panda. Performance comparison of mpi implementations over infini-band, myrinet and quadrics. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 58. IEEE Computer Society, 2003.
- [124] Rita Loogen, Yolanda Ortega, Ricardo Pe na, Steffen Priebe, and Fernando Rubio. *Parallelism abstractions in eden*, pages 95–128. Springer-Verlag, London, UK, 2003.
- [125] F. Loulergue. Implementation of a functional bulk synchronous parallel programming library. In *14th IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.
- [126] Frédéric Loulergue, Gaétan Hains, and Christian Foisy. A calculus of functional bsp programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.
- [127] Joan D. Lukas, Kenneth Newman, and John P. Sullivan. A data-parallel extension to split-c. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, USA, Mars 1997.
- [128] Craig Lund. Altivec Introduction. Publication commerciale de Motorola, 2001.
- [129] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.
- [130] B.L. Massingill, T.G. Mattson, and B.A. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999)*, 1999.
- [131] Clément Ménier, Edmond Boyer, and Bruno Raffin. 3d skeleton-based body pose recovery. In *Proceedings of the 3rd International Symposium on 3D Data Processing, Visualization and Transmission*, june 2006.
- [132] H. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon. Clusters top 500. <http://clusters.top500.org>, 2005.
- [133] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 28(8):114–117, 1965.

- [134] Hans Moravec. Visual mapping by a robot rover. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, pages 599–601, August 1979.
 - [135] C. Morimoto. *Electronic Digital Stabilization: Design and Evaluation, with Applications*. PhD thesis, University of Mariland, 1997.
 - [136] T. N. Mundhenk, C. Ackerman, D. Chung, N. Dhavale, B. Hudson, R. Hirata, E. Pichon, Z. Shi, A. Tsui, and L. Itti. Low-cost high-performance mobile robot design utilizing off-the-shelf parts and the beowulf concept: the beobot project. In *Proc. SPIE Conference on Intelligent Robots and Computer Vision XXI: Algorithms, Techniques, and Active Vision*, pages 293–303. SPIE Press, Oct 2003.
 - [137] Nathan Myers. A new and useful template technique: traits. *C++ gems*, 1:451–457, 1996.
 - [138] A. Niculescu-Mizil and R. Caruana. Obtaining calibrated probabilities from boosting. In *Proc. 21st Conference on Uncertainty in Artificial Intelligence (UAI '05)*. AUAI Press, 2005.
 - [139] D. Nistér. Automatic dense reconstruction from uncalibrated video sequences. In *European Conference on Computer Vision*, pages 649–663, 2000.
 - [140] Katja Nummiaro, Esther Koller-Meier, Tomáš Svoboda, Daniel Roth, and Luc Van Gool. Color-based object tracking in multi-camera environments. In B. Michaelis and G. Krell, editors, *25th Pattern Recognition Symposium, DAGM'03*, number 2781 in LNCS, pages 591–599. Springer, September 2003.
 - [141] Marc Olano and Anselmo Lastra. A shading language on graphics hardware: the pixelflow shading system. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 159–168, New York, NY, USA, 1998. ACM Press.
 - [142] Ian Ollman. Altivec : A Velocity Engine Tutorial. Ressource en ligne – <http://wwwsimdtech.org/altivec>, 2003.
 - [143] M. Oren, C.P. Papageorgiou, P. Sinha, E. Osuna, and T. Poggio. Pedestrian detection using wavelet templates. In *Proceedings of CVPR'97*, pages 193–99, 1997.
 - [144] D.V. Papadimitriou and T.J. Dennis. Epipolar line estimation and rectification for stereo image pairs. *T-IP*, 5:672–676, 1996.
-

- [145] PARISYS. The parsys transalpha cross development toolset - reference manual. Technical report, PARISYS Ltd., Grande-Bretagne, Avril 1996. Doc. v2.0 (réf. 2180-0100).
 - [146] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.
 - [147] E. Pichon and L. Itti. Real-time high-performance attention focusing for outdoors mobile beobots. In *AAAI Spring Symposium*, page 63, Mar 2002.
 - [148] M. Pollefeys, R. Koch, and L. Van Gool. Self-calibration and metric reconstruction in spite of varying and unknown internal camera parameters0. In *International Conference on Computer Vision*, pages 90–95, 1998.
 - [149] P.A. Revenga, J. Serot, J. L. Lazaro, and J.P. Derutin. A beowulf-class architecture proposal for real-time embedded vision. In *International Parallel and Distributed Processing Symposium*, 2003.
 - [150] Calvin J. Ribbens, Dennis Kafura, Amit Karnik, and Markus Lorch. The virginia tech computational grid: A research agenda.
 - [151] Guillaume Rincé. La Technologie AltiVec et sa mise en oeuvre au sein du Motorola MPC 7410. Publication interne de l’ENSTA, 2001.
 - [152] D. Szafron S. MacDonald and J. Schaeffer. Object-oriented pattern-based parallel programming with automatically generated frameworks. In *Proceedings of COOTS’99*, pages 29–43, 1999.
 - [153] Hideo Saito, Shigeyuki Baba, Makoto Kimura, Sundar Vedula, and Takeo Kanade. Appearance-based virtual view generation of temporally-varying events from multi-camera images in the 3d room. In *Proceedings of Second International Conference on 3-D Digital Imaging and Modeling*, pages 516–525, 1999.
 - [154] Robert E. Schapire. A brief introduction to boosting. In *IJCAI*, pages 1401–1406, 1999.
 - [155] Julien Sebot and Nathalie Drach-Temam. Memory bandwidth: The true bottleneck of SIMD multimedia performance on a superscalar processor. *Lecture Notes in Computer Science*, 2150:439–??, 2001.
 - [156] Frank J. Steinstra. *User Transparnt Parallel Image Processing*. PhD thesis, Université d’Amsterdam, 2003.
 - [157] Jocelyn Sérot and Dominique Ginhac. Skeletons for parallel image processing: an overview of the skipper project. *Parallel Computing*, 28(12):1685–1708, 2002.
-

- [158] Jocelyn Sérot, Dominique Ginhac, and Jean-Pierre Dérutin. Skipper: a skeleton-based parallel programming environment for real-time image processing applications. In V. Malyshkin, editor, *5th International Conference on Parallel Computing Technologies (PaCT-99)*, volume 1662 of *LNCS*, pages 296–305. Springer, 6–10 September 1999.
 - [159] Leo Chin Sim, Heiko Schröder, and Graham Leedham. Mimd-simd hybrid system—towards a new low cost parallel system. *Parallel Computing*, 29(1):21–36, 2003.
 - [160] Ajit Singh, Jonathan Schaeffer, and Duane Szafron. Experience with parallel programming using code templates. *Concurrency: Practice and Experience*, 10(2):91–120, 1998.
 - [161] S. Siu, M. De Simone, D. Goswami, and A. Singh. Design patterns for parallel programming. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pages 230–240, 1996.
 - [162] David B. Skillicorn. A taxonomy for computer architectures. *Computer*, 21(11):46–57, 1988.
 - [163] David B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, 1990.
 - [164] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.
 - [165] Nathan Slingermaier and Allan Jay Smith. Performance Analysis of Instruction Set Architecture Extensions for Multimedia. Technical report, Apple Computer Inc. and University of California at Berkeley, 2001.
 - [166] M. Smit, J. Garegnani, M. Bechdol, and S. Chettri. Parallel image classification on the hive. *29th Applied Imagery Pattern Recognition Workshop*, 00:39, 2000.
 - [167] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, volume 1, pages 11–14, 1995.
 - [168] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese. *How to build a Beowulf: a guide to the implementation and application of PC clusters*. MIT Press, 1999.
 - [169] Erich Strohmaier, Jack J. Dongarra, Hans W. Meuer, and Horst D. Simon. Recent trends in the marketplace of high performance computing. *Parallel Comput.*, 31(3+4):261–273, 2005.
-

- [170] X.-H. Sun and L. M. Ni. Another view on parallel speedup. In *Supercomputing '90*, pages 324–333, 1990.
 - [171] S. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.
 - [172] T. Svoboda, H. Hug, and L. Van Gool. Viroom — low cost synchronized multicamera system and its self-calibration, 2002.
 - [173] Tomáš Svoboda, Hanspeter Hug, and Luc Van Gool. ViRoom — low cost synchronized multicamera system and its self-calibration. In Luc Van Gool, editor, *Pattern Recognition, 24th DAGM Symposium*, number 2449 in LNCS, pages 515–522. Springer, September 2002.
 - [174] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.
 - [175] Yoshio Tanaka, Motohiko Matsuda, Makoto Ando, Kazuto Kubota, and Mitsuhsisa Sato. Compas: A pentium pro PC-based SMP cluster and its experience. In *IPPS/SPDP Workshops*, pages 486–497, 1998.
 - [176] The BlueGene/L Team. An overview of the bluegene/l supercomputer. In *Proceddings of ACM Supercomputing Conference*, 2002.
 - [177] V. Teulière and Olivier Brun. Parallelisation of the particle filtering technique and application to doppler-bearing tracking of maneuvering sources. *Parallel Comput.*, 29(8):1069–1090, 2003.
 - [178] K. Tieu and P. Viola. Boosting image retrieval. *International Journal of Computer Vision*, 56(1):17–36, 2004.
 - [179] Dave Turner and Xuehua Chen. Protocol-dependent message-passing performance on linux clusters. *IEEE International Conference on Cluster Computing*, 00:187 – 194, 2002.
 - [180] Dave Turner, Adam Oline, Xuehua Chen, and Troy Benjegerdes. Integrating new capabilities into netpipe. In *PVM/MPI*, pages 37–44, 2003.
 - [181] E. Unruh. Prime number computation. Technical Report ANSI X3J16-94-0075/ISO WG21-462., C++ Standard Committee, 1994.
 - [182] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
 - [183] Sundar Vedula, Simon Baker, and Takeo Kanade. Image-based spatio-temporal modeling and view interpolation of dynamic events. *ACM Transactions on Graphics*, 24:240 – 261, 2005.
-

- [184] Sundar Vedula, Simon Baker, Peter Rander, Robert Collins, and Takeo Kanade. Three-dimensional scene flow. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:475 – 480, 2005.
 - [185] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
 - [186] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
 - [187] Todd L. Veldhuizen. C++ templates are turing complete.
 - [188] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.
 - [189] Todd L. Veldhuizen. Arrays in blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
 - [190] Todd L. Veldhuizen. C++ templates as partial evaluation. In O. Danvy, editor, *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18, San Antonio, January 1999. University of Aarhus, Dept. of Computer Science.
 - [191] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Philadelphia, PA, USA, 1998. SIAM.
 - [192] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 511–518, 2001.
 - [193] John von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.
 - [194] R. Weper, E. Zehendner, and W. Erhard. P: Hierarchical modeling of parallel architectures. *Seventh Euromicro Workshop on Parallel and Distributed Processing*, 0:233, 1999.
 - [195] Ruigang Yang and Marc Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware. In *CVPR (1)*, pages 211–220, 2003.
 - [196] Ruigang Yang and Marc Pollefeys. A versatile stereo implementation on commodity graphics hardware. *Real-Time Imaging*, 11(1):7–18, 2005.
-