Algorithmes parallèles de simulation physique pour la synthèse d'images : application à l'animation de textiles

Florence Zara

To cite this version:

Florence Zara. Algorithmes parallèles de simulation physique pour la synthèse d'images : application à l'animation de textiles. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 2003. Français. NNT : . tel-00005117v2

HAL Id: tel-00005117
https://theses.hal.science/tel-00005117v2

Submitted on 5 Mar 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° attribué par la bibliothèque
|_|_|_|_|_|_|_|_|_|_|_|

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : "Informatique : Systèmes et Logiciels"

préparée au laboratoire INFORMATIQUE ET DISTRIBUTION
et en coopération avec le laboratoire GRAPHISME, VISION ET ROBOTIQUE
dans le cadre de ***l'École Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique"***

présentée et soutenue publiquement

par

Florence ZARA

le 1er décembre 2003

Algorithmes parallèles de simulation physique pour la synthèse d'images : application à l'animation de textiles

---

Directeur de thèse : Mme Brigitte Plateau

JURY

MME MARIE-PAULE CANI , Présidente
M. SERGE MIGUET , Rapporteur
M. JEAN-MICHEL DISCHLER , Rapporteur
MME BRIGITTE PLATEAU , Examinateur (directeur de thèse)
M. JEAN-MARC VINCENT , Examinateur (co-encadrant)
M. FRANÇOIS FAURE , Examinateur (co-encadrant)

A mes parents,
Bernard et Marie-Madeleine,
et à mon frère,
Philippe.

Remerciements

Je remercie Mme Marie-Paule Cani, professeur à l'INPG, qui m'a fait l'honneur de présider ce jury. Je remercie également M. Jean-Michel Dischler, professeur à l'Université Louis Pasteur de Strasbourg et M. Serge Miguet, professeur à l'Université Lyon 2 d'avoir bien voulu accepter la charge de rapporteur. Je remercie de même Mme Brigitte Plateau, professeur à l'INPG, pour avoir acceptée d'être officiellement ma directrice de thèse. Et je remercie enfin M. Jean-Marc Vincent et M. François Faure, maîtres de conférences à l'Université Joseph Fourier, pour m'avoir co-encadrée durant ma thèse.

Voici la dernière ligne droite au bout de laquelle mes années de doctorat s'achèvent. Cette dernière étape permet de remercier enfin à leurs justes valeurs l'ensemble des personnes sans lesquelles ces années de recherche n'auraient sans doute pas eu la même saveur.

En premier lieu, je tiens depuis très longtemps à remercier Titou et Rémi qui ont passé de nombreuses heures, soirées et week-end à faire en sorte qu'ATHAPASCAN fonctionne de mieux en mieux. De nombreuses pizza furent mangées, de nombreuses heures de sommeil sacrifiées et les week-end oubliés...

Ensuite une seconde équipe de développeurs a fait son apparition au laboratoire ID, apportant dans leurs bagages joysticks et vidéo-projecteurs ainsi qu'un nouveau challenge identifié sous le nom de SIGGRAPH. Je parle bien entendu de Bruno dont sa maîtrise parfaite de l'anglais, à défaut des scies japonaises, m'a beaucoup aidée ; de Jérémie qui sait en quelques secondes planter un décor, faire couler une rivière et sortir un lapin de son chapeau ; et enfin de Loïck toujours présent pour m'expliquer les nouveautés apportées durant la nuit sur la fameuse "démo valley" désormais mondialement connue...

Par ailleurs, quand on est une jeune demoiselle informaticienne et que la joie de recompiler son noyau Linux tous les soirs avant de se coucher nous est complètement inconnue, mieux vaut connaître un informaticien qui le fait toujours avec un immense bonheur. C'est pourquoi je tiens à remercier Cyrille mon administrateur personnel pour toute son aide, pour les soirées passées autour d'un verre et pour les nombreuses personnes qu'il m'a faites connaître.

Je voulais également dire à Olivier R. que toutes ses discussions philosophiques, spirituelles ou encore en rapport avec un livre, une BD ou un film vont beaucoup me manquer. Bon courage pour ces prochains mois qui vont être rudes au niveau du sommeil mais ô combien remplis de bonheurs nouveaux qui feront sans doute immédiatement oublier toutes les cernes accumulées.

Merci également à Corine pour m'avoir soutenue quand je n'avais pas le moral et pour comprendre ce que je ressentais à certains moments de ma thèse ; à Simon et Stéphane M. pour leurs prouesses en ce qui concerne l'administration de la grappe et leurs patiences lors de mes premiers essais peu appréciés par le cluster ; à Jul. et Adrien pour m'avoir à de

nombreuses reprises fait oublier mon stress par leurs conneries sans cesse renouvelées ; à Stéphane G. de mon autre laboratoire pour toute son aide apportée en OpenGL et en relecture d'articles ainsi que pour m'avoir fait découvrir une autre culture et de nouveaux paysages ; et enfin à Nico, infographiste indépendant, pour mon beau site Web, ses belles idées, et les soirées improvisées...

Et pour rythmer toutes ces heures de travail acharnées, j'avais deux secrets nommés Têtes Raides et Mark XIII Dept. 1 (merci Aymeric ! ! !).

Enfin sans toutes les nommer explicitement merci également à tous les autres membres du laboratoire ID et des équipes EVASION et ARTIS du laboratoire GRAVIR ; merci à mes deux chefs de laboratoire : Brigitte toujours de bonne humeur malgré ses problèmes de véranda et une équipe à encadrer et Marie-Paule dont les croquis durant les conférences valent le coup d'œil.

Et enfin un grand merci à tous mes amis qui ne m'ont malheureusement pas beaucoup vue cette dernière année ; merci à Six pour avoir subi mes défoulements au squash ; et merci "aux filles", en particulier à MC et Fab., pour toutes nos soirées bien arrosées d'éclats de rire en autres choses...

Pour finir un grand merci à mon Yéyé qui m'a soutenue, mise en confiance et supportée durant ces durs mois de fin de thèse...

Table des matières

1 Introduction **1**

Problématique . . . 1

Contributions . . . 4

Organisation du manuscrit . . . 4

I La simulation de textiles **7**

2 Modèles physiques **11**

2.1 Introduction . . . 11

2.2 Modèles discrets . . . 12

2.2.1 Systèmes masses-ressorts de Provot . . . 14

2.2.2 Modèle de Kawabata . . . 16

2.2.3 Formulation de l'équation du mouvement . . . 17

2.3 Modèles continus . . . 24

2.3.1 Équation du mouvement . . . 24

2.3.2 Calcul des déformations . . . 25

2.3.3 Résolution de l'équation du mouvement . . . 25

2.4 Détection et traitement des collisions . . . 26

2.5 Conclusion . . . 27

3 Méthodes d'intégration numérique **31**

3.1 Introduction . . . 31

3.2 Forme abstraite du système différentiel . . . 33

3.3 Méthode d'Euler explicite . . . 33

3.3.1 Schéma d'intégration numérique . . . 33

3.3.2 Interprétation géométrique . . . 34

3.3.3 Convergence . . . 34

3.3.4 Stabilité . . . 35

3.3.5 Application à la simulation de textiles . . . 37

3.4 Méthode de Stöermer-Verlet/leapfrog . . . 37

3.4.1 Schéma d'intégration numérique 37
3.4.2 Stabilité 38
3.4.3 Application à la simulation de textiles 39
3.5 Méthode d'Euler implicite 40
3.5.1 Schéma d'intégration numérique 40
3.5.2 Stabilité 40
3.5.3 Application à la simulation de textiles 41
3.6 Conclusion 45

4 Sources de parallélisme **47**
4.1 Introduction 47
4.2 Boucle de la simulation 48
4.2.1 Calcul des forces 48
4.2.2 Calcul des accélérations 49
4.2.3 Schémas d'intégration explicites 50
4.2.4 Schéma d'intégration implicite 50
4.3 Conclusion 54

II Parallélisme **57**

5 Calcul parallèle **61**
5.1 Introduction 61
5.2 Conception d'algorithmes parallèles 62
5.2.1 Méthodologie 62
5.2.2 Partitionnement 64
5.2.3 Communication 66
5.2.4 Agglomération 67
5.2.5 Ordonnancement 70
5.2.6 Interaction 71
5.3 Modèles de performance 72
5.3.1 Temps d'exécution 72
5.3.2 Accélération 72
5.3.3 Efficacité 72
5.3.4 Coût 73
5.4 Conclusion 73

6 Architectures parallèles **75**
6.1 Introduction 75
6.2 Architectures parallèles 76
6.2.1 Machines à mémoire partagée 77
6.2.2 Machines à mémoire distribuée 78

6.2.3 Machines graphiques parallèles 79
6.2.4 Grappes de machines ou clusters 79
6.3 Modèles de programmation parallèle 81
6.3.1 Parallélisme de données 82
6.3.2 Parallélisme de contrôle 82
6.4 Environnements de programmation parallèle 83
6.4.1 Échange de messages : PVM et MPI 83
6.4.2 Mémoire partagée : Cilk 86
6.4.3 Mémoire virtuelle partagée : ATHAPASCAN 87
6.5 Conclusion 88

7 ATHAPASCAN 91
7.1 Introduction 91
7.2 Programme ATHAPASCAN 93
7.3 Tâches et objets partagés 93
7.3.1 Tâche ATHAPASCAN 93
7.3.2 Procédure associée à une tâche 94
7.3.3 Création et exécution d'une tâche 94
7.3.4 Droits d'accès des données partagées 95
7.3.5 Accès mémoire possibles pour une tâche 96
7.4 Modèle d'exécution 97
7.4.1 Garanties fournies par ATHAPASCAN 97
7.4.2 Phases d'exécution 97
7.4.3 Construction du graphe de flots de données 98
7.4.4 Support d'exécution 99
7.5 Conclusion 100

III Parallélisation et couplage de programmes parallèles 101

8 Partitionnement 105
8.1 Introduction 105
8.2 Algorithme parallèle du Gradient Conjugué 106
8.2.1 Références bibliographiques 106
8.2.2 Algorithme parallèle 107
8.3 Simulations parallèles 109
8.3.1 Simulation de vêtements : Romero *et al.* 109
8.3.2 Dynamique moléculaire : Bernard *et al.* 111
8.4 Partitionnement 112
8.4.1 Décomposition de l'espace objet 113
8.4.2 Tâches associées aux fragments de données 118

8.5 Conclusion 119

9 Création et exécution des tâches 121

9.1 Introduction 121
9.2 Création des tâches 122
9.2.1 Représentation des données 122
9.2.2 Calcul des forces 123
9.2.3 Calcul des accélérations 126
9.2.4 Schémas d'intégration explicites 127
9.2.5 Schéma d'intégration implicite 129
9.3 Ordonnancement 143
9.3.1 Contrôle de l'ordonnanceur 143
9.3.2 Annotations de code 144
9.3.3 Algorithmes d'ordonnancement de graphes 146
9.4 Conclusion 147

10 Évaluation de performances 149

10.1 Introduction 149
10.2 Complexité des algorithmes parallèles 150
10.2.1 Nombre de tâches créées 150
10.2.2 Complexités pratiques 152
10.2.3 Temps d'exécution des tâches 153
10.2.4 Potentiel de parallélisme 155
10.3 Mémoire utilisée 156
10.4 Influence de la granularité 157
10.5 Influence de l'ordonnancement 160
10.5.1 Placement du GFD explicite 160
10.5.2 Placement du GFD implicite 165
10.5.3 Temps de l'exécution parallèle 169
10.6 Conclusion 173

11 Couplage de programmes parallèles 175

11.1 Introduction 175
11.2 Réalité Virtuelle 177
11.2.1 Caractéristiques 177
11.2.2 Dispositifs d'affichage 178
11.2.3 Logiciel de Réalité Virtuelle 179
11.3 Réalité Virtuelle sur cluster 179
11.3.1 Caractéristiques 179
11.3.2 Environnements de RV sur cluster 180
11.4 Couplage des programmes parallèles 182

11.4.1 Visualisation 182
11.4.2 Couplage 186
11.4.3 Améliorations 190
11.5 Conclusion 191

12 Conclusion et perspectives 193

IV Annexes 197

A Méthode du Gradient Conjugué 199
A.1 Introduction 199
A.2 Méthode du Gradient Conjugué 200
A.2.1 Forme quadratique 201
A.2.2 La méthode de la plus grande pente 203
A.2.3 La méthode des directions conjuguées 204
A.2.4 La méthode des gradients conjugués 206
A.3 Algorithmes du Gradient Conjugué 208
A.3.1 Algorithme séquentiel 208
A.3.2 Algorithme parallèle 209
A.4 Conclusion 211

B Galerie d'images 213

Bibliographie 217

Publications 228

Résumé 230

Table des figures

2.1 Discrétisation du textile en un maillage polygonal. Les sommets sont appelés particules. Sa topologie définit les interactions et les forces entre les particules. 13
2.2 Le système masses-ressorts de Provot avec (1) les ressorts longitudinaux, (2) les ressorts diagonaux et (3) les ressorts de flexion 14
2.3 Courbe de Kawabata présentant la flexion d'un tissu 100% coton 16
2.4 Forces appliquées sur une particule : forces dues aux ressorts (forces locales) et forces extérieures (gravité, vent : forces globales) 18
2.5 Forme des matrice des dérivées des forces $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial v}$. Les éléments diagonaux sont non nuls. Ils correspondent à la somme des contributions des forces de tous leurs voisins. Les éléments non nuls en dehors de la diagonale représentent leurs propres contributions par rapport à la particule de la diagonale. 20
2.6 La configuration de référence : l'état d'équilibre est défini par le déplacement **r** dans l'espace $U \times V$. Il est ensuite transformé par la déformation **d** en une nouvelle configuration (déformée), définie par le déplacement **s** dans l'espace $U \times V$. 25
2.7 Exemple d'utilisation d'un maillage adaptatif lors du traitement des collisions (images du projet Modeling of Virtual Textiles (MoViTex), extraites de leur site Web : http ://www.gris.uni-tuebingen.de/projects/physsim/). . 28

3.1 Interprétation géométrique du schéma d'intégration d'Euler explicite : l'approximation de la solution au temps $t + h$ est obtenue en calculant la tangente à la solution au temps t. 34
3.2 Solutions du système différentiel défini par $y'(t) = \lambda y(t), y(0) = 1$ avec $\lambda = -15$ et $\lambda = 2$, pour $t \in [0, 1]$ 35
3.3 Région de stabilité de la méthode d'Euler explicite pour l'équation $y'(t) = \lambda y(t)$ 36
3.4 Grilles échelonnées pour la méthode d'intégration de Störmer-Verlet/leapfrog 37
3.5 Région de stabilité de la méthode d'Euler implicite pour l'équation $y'(t) = \lambda y(t)$ 41

4.1 Forme de la matrice $\frac{df}{dx}$ associé au maillage triangulaire des 15 particules dans le plan 2D : les éléments diagonaux sont non nuls et correspondent à $\frac{df_i}{dx_i}$, les éléments en dehors de la diagonale non nuls correspondent à $\frac{df_i}{dx_j}$ avec j voisin de la particule i sachant que $\frac{df_i}{dx_j} = \frac{df_j}{dx_i}$ 52

5.1 Méthodologie pour la conception de programmes parallèles. Le problème initial est partitionné en sous problèmes ; les communications nécessaires sont établies ; les sous problèmes sont ensuite agglomérés en tâches ; ces tâches sont ensuite placées sur les processeurs ; enfin les interactions entre la simulation et la visualisation sont gérées. 63

5.2 Décomposition de l'espace pour un problème impliquant une grille à trois dimensions. Des décompositions à une-, deux- ou trois-dimensions sont possibles. Dans la majorité des cas, la décomposition en trois dimensions est privilégiée, offrant le plus de flexibilité. 65

5.3 Décomposition de l'espace pour un problème impliquant une grille à 3 dimensions. Une décomposition en trois dimensions a été privilégiée. Des tâches adjacentes ont été combinées permettant l'augmentation de la granularité initialement obtenue. 68

5.4 Partitionnement 2D fin d'un calcul établi en chacun des points d'une grille 8×8 avec 64 tâches. Les communications entrantes (claires) et sortantes (foncées) d'une tâche donnée sont détaillées. 69

5.5 Partitionnement 2D grossier d'un calcul établi en chacun des points d'une grille 8×8 avec 4 tâches. Les communications entrantes (claires) et sortantes (foncées) d'une tâche donnée sont détaillées. 69

5.6 Placement d'un problème de calcul basé sur une grille. Chaque tâche effectue la même quantité de calcul et ne communique qu'avec ces quatre voisins. Les lignes épaisses en pointillés délimitent les frontières des processeurs. 70

6.1 Machine multiprocesseurs à mémoire partagée. P représente un processeur indépendant. 77

6.2 Modélisation d'une machine parallèle MIMD à mémoire distribuée. Chaque noeud est composé d'un processeur et d'une mémoire locale. Les noeuds communiquent entre eux en envoyant et en recevant des messages via le réseau de communication. 78

6.3 Architecture du système SGI Origin. Deux processeurs sont connectés via un “hub” à une mémoire partagée pour former un “noeud”. Plusieurs noeuds sont ensuite connectés en utilisant un réseau d'interconnexion pour former le système global. 79

6.4 (A gauche) Grappe de PCs du projet iCluster (HP, INRIA, ID-IMAG) composée de 226 mono-processeurs Pentium III à 733 MHz avec 256 Mo de mémoire, inter-connectés via un réseau de 100 Mbit/s . (A droite) Grappe du projet iCluster2 composée de 104 bi-processeurs Itanium-2, 64 bits à 900 MHz avec 3 Go de mémoire, inter-connectés par un réseau Myrinet. 80

7.1 Graphe de flots de données de la simulation de textiles. Les rectangles, les losanges et les cercles représentent respectivement les accès aux données partagées, les données partagées et les tâches de calculs. Les traits représentent les dépendances. 92
7.2 Graphe de flots de données de la simulation de textiles 98

8.1 Coefficients de la matrice originale à gauche. Après une réorganisation MRD (Multiple Recursive Distribution) au milieu. Et après une réorganisation en bandes à droite. 110
8.2 Distribution originale à gauche. Après une réorganisation des particules au milieu. Et après une réorganisation des triangles à droite. 110
8.3 Découpage spatial de l'espace de simulation dans le cadre d'une application de dynamique moléculaire 112
8.4 Décomposition du tissu en sous ensembles de particules de dimension `DimBox` 113
8.5 Structures de données reflétant le partitionnement de l'objet simulé : à gauche le tableau contient l'ensemble des particules, à droite ce tableau a été découpé en blocs contenant des sous-ensembles de particules 114
8.6 Courbes récursives de Hilbert (en haut) et de Peano (en bas) pour $n = 0, ..., 3$ 115
8.7 Communications issues du partitionnement du tissu en blocs de particules : cas où chacun des blocs se trouve sur un processeur différent . . . 116
8.8 Optimisation des communications issues du partitionnement : seules les frontières des blocs sont transmises aux processeurs ayant les blocs voisins 117
8.9 Simulation de n particules sur p processeurs avec un partitionnement en une grille de k blocs 118

9.1 Graphe de flots de données associé aux calculs des forces relatives au bloc i. Les particules contenues dans le $i^{ème}$ bloc ont des interactions avec celles disposées dans les blocs j_1 et j_2 (tâches de calcul T_{j_1} et T_{j_2}), auxquelles il faut rajouter les interactions locales au bloc i (tâche de calcul T_i). 125
9.2 Graphe de flots de données associé aux calculs des accélérations des particules contenues dans le bloc i. Pour calculer leurs accélérations, il suffit de connaître leurs forces, ainsi que leurs masses. Aucune information relative aux autres blocs n'est nécessaire. 127

9.3 Graphe de flots de données associé aux calculs des positions et vitesses relatives au bloc i via un schéma d'intégration explicite. Pour effectuer ces calculs, il suffit de connaître les états (position, vitesse et accélération) des particules de ce même bloc. Aucune information relative aux autres blocs n'est nécessaire. 128
9.4 Graphe de flots de données associé à une itération de la simulation de textiles dans le cas de l'utilisation d'une méthode d'intégration explicite . 129
9.5 Structures de données associées aux matrices des contributions des forces. Les éléments diagonaux sont séparés des éléments non diagonaux non nuls. Les deux structures employées sont fragmentées en blocs. 130
9.6 Graphe de flots de données associé aux calculs des contributions des forces relatives au bloc i. Les particules contenues dans le $i^{ème}$ bloc ont des interactions avec celles disposées dans les blocs j_1 et j_2. Mais à ces interactions il ne faut pas oublier de rajouter les interactions locales au bloc i. 133
9.7 Première étape du calcul en parallèle du vecteur b. Deux types de tâches sont créées. Les premières effectuent la multiplication des éléments diagonaux de $\frac{\partial f}{\partial x}$ par le vecteur $v(t)$. Les secondes effectuent la même multiplication mais avec les éléments non diagonaux de la matrice. Les résultats sont additionnés au fur et à mesure dans le vecteur b. 137
9.8 Graphe de flots de données associé au calcul du bloc i du vecteur b, multiplication des éléments de la matrice des contributions avec le vecteur vitesse. Deux types de tâches permettent l'obtention du résultat de ce calcul. Les premières multiplient les éléments diagonaux de la matrice au vecteur $v(t)$, tandis que les secondes traitent les éléments non diagonaux de la matrice. 138
9.9 Graphe de flots de données associé à la dernière tâche de calculs des éléments du bloc `B[i]`. Aucune information relative aux autres blocs n'est nécessaire. 139
9.10 Graphe de flots de données associé à la mise à jour des positions et vitesses relatives au bloc i via un schéma d'intégration implicite. Pour effectuer ces calculs, il suffit de connaître les positions et vitesses des particules de ce même bloc, ainsi que le bloc solution Δv correspondant. Aucune information relative aux autres blocs n'est nécessaire. 142
9.11 Partitionnement du graphe de dépendances de la simulation de textiles . . 147

10.1 Tissu régulier de taille 100×100 partitionné en 4 blocs identiques de 2 500 particules 150
10.2 Simulation de 10 000 particules avec la méthode d'intégration de Leapfrog : influence de la découpe sur le nombre de tâches créées et sur le temps d'exécution sur 1 noeud 158

10.3 Simulation de 10 000 particules avec la méthode d'intégration d'Euler implicite : influence de la découpe sur le nombre de tâches créées et sur le temps d'exécution sur 1 noeud 159
10.4 Graphe de flots de données associé à la simulation lors de l'emploi d'une méthode d'intégration explicite 161
10.5 Graphe de flots de données associé à la simulation partitionnée en 4 sous-ensembles de particules lors de l'emploi d'une méthode d'intégration explicite 161
10.6 Placement Scotch (à gauche) et Cyclic (à droite) du graphe de flots de données explicite sur 5 itérations pour un partitionnement en 4 sous-ensembles de particules sur 2 processeurs 162
10.7 Placement Scotch (à gauche) et Cyclic (à droite) du graphe de flots de données explicite sur 5 itérations pour un partitionnement en 4 sous-ensembles de particules sur 4 processeurs 163
10.8 Partitionnement Scotch du graphe de flots de données explicite sur 5 itérations pour un découpage en 4 sous-ensembles de particules sur 2 processeurs 164
10.9 Partitionnement Scotch du graphe de flots de données explicite sur 5 itérations pour un découpage en 4 sous-ensembles de particules sur 4 processeurs 164
10.10 Graphe de flots de données associé à la simulation lors de l'emploi de la méthode d'intégration d'Euler implicite 165
10.11 Graphe de flots de données associé à la simulation partitionnée en 4 sous-ensembles de particules lors de l'emploi de la méthode d'Euler implicite . 166
10.12 Placement Scotch (à gauche) et Cyclic (à droite) du GFD implicite sur 2 itérations avec 1 itération du GC pour un partitionnement en 4 sous-ensembles de particules sur 2 processeurs 167
10.13 Placement Scotch (à gauche) et Cyclic (à droite) du GFD implicite sur 2 itérations avec 1 itération du GC pour un partitionnement en 4 sous-ensembles de particules sur 4 processeurs 168
10.14 Performances obtenues pour une itération d'un problème comportant 490000 particules avec une taille de blocs de 2 100 particules en employant la méthode d'intégration numérique de Stöermer-Verlet/Leapfrog sur le *I-Cluster* 170
10.15 Performances obtenues pour une itération d'un problème comportant 1 000 000 particules avec une taille de blocs de 2 000 particules en employant la méthode d'intégration numérique de Stöermer-Verlet/Leapfrog sur la grappe *ARV* 170
10.16 Performances obtenues pour une itération d'un problème comportant 490000 particules avec une taille de blocs de 2 500 particules en employant la méthode d'intégration numérique de Stöermer-Verlet/Leapfrog sur le *I-Cluster* 171
10.17 Performances obtenues pour une itération d'un problème comportant 490000 particules avec une taille de blocs de 2 500 particules en employant la méthode d'intégration d'Euler implicite sur le *I-Cluster* 172

11.1 Système immersif multiprojection de Réalité Virtuelle appelé CAVE . . . 178
11.2 Illustration de VR Juggler Cluster dans le cadre de la méthode hybride. L'application VR Juggler est distribuée sur deux noeuds et les primitives graphiques OpenGL sont également distribuées à l'aide de WireGL sur 4 autres noeuds du cluster. 180
11.3 Illustration de Net Juggler. L'application est exécutée sur chacun des noeuds. Seul un noeud reçoit les données en entrées qu'il diffuse ensuite vers tous les autres noeuds afin d'assurer la cohérence des données. 181
11.4 Représentation d'un morceau de tissu. Le textile est discrétisé en un maillage triangulaire de particules. Chacun des triangles est appelé facette. Le tissu peut être affiché en mode filaire (à gauche) ou bien il est possible de lui appliqué une texture (à droite). 183
11.5 Visualisation de la simulation de textiles sur trois écrans via l'environnement de réalité virtuelle multi-projecteurs Net Juggler 186
11.6 Couplage de la simulation parallèle (ATHAPASCAN) avec une visualisation multi écrans (Net Juggler). Les deux parties de l'application s'exécutent sur des grappes de machines différentes. 187
11.7 Couplage de la simulation parallèle (ATHAPASCAN) avec une visualisation multi écrans (Net Juggler). Les deux parties de l'application s'exécutent sur la même grappe de machines. 188
11.8 Interaction de la simulation de textiles avec l'utilisateur. Le coin en haut à gauche est fixé, tandis que le coin en haut à droite est dirigé par l'utilisateur via la souris. 189
11.9 Liaison TCP de type N-N entre la grappe de simulation et celle de visualisation 191

A.1 Système linéaire à deux dimensions : la solution du système correspond au point d'intersection des deux droites 201
A.2 Graphe et courbes de niveau de la forme quadratique $f(x)$: le point minimum de cette surface est la solution de $Ax = b$, chaque courbe ellipsoïdale représente un $f(x)$ constant, le point correspond au $f(x)$ minimum . 202
A.3 Méthode de la plus grande pente commençant à $x_{(0)} = [-2, -2]^T$ et convergeant pour $x = [2, -2]^T$ 204
A.4 La méthode des directions conjuguées 205
A.5 La méthode des Gradients Conjugués 207

Liste des algorithmes

1 Boucle de la simulation de textiles . . . 28
2 Algorithme du Gradient Conjugué pour la résolution de $Ax = b$ . . . 44
3 Boucle de la simulation de textiles . . . 48
4 Calcul des forces exercées sur les particules . . . 49
5 Calcul des accélérations des particules . . . 49
6 Intégration via un schéma explicite (Euler ou Stöermer-Verlet/leapfrog) . 50
7 Intégration via le schéma d'Euler implicite (étape 1) . . . 51
8 Intégration via le schéma d'Euler implicite (étape 3) . . . 53
9 Intégration via le schéma d'Euler implicite (étape 4) . . . 54
10 Définition de la procédure associée à une tâche ATHAPASCAN . . . 94
11 Exécution de la procédure associée à une tâche ATHAPASCAN . . . 95
12 Calcul parallèle des forces exercées sur les particules . . . 124
13 Calcul parallèle des accélérations des particules . . . 126
14 Intégration en parallèle via un schéma explicite . . . 128
15 Intégration en parallèle via le schéma d'Euler implicite (étape 1) . . . 131
16 Intégration en parallèle via le schéma d'Euler implicite (étape 2) : remplissage du vecteur `B[]` à partir des éléments diagonaux de la matrice $\frac{\partial f}{\partial x}$ . . . 135
17 Intégration en parallèle via le schéma d'Euler implicite (étape 2) : remplissage du vecteur b à partir des éléments non diagonaux de la matrice . 136
18 Intégration en parallèle via le schéma d'Euler implicite (étape 2) : phase finale du remplissage du vecteur `B[]` . . . 138
19 Algorithme simplifié du Gradient Conjuguée . . . 140
20 Intégration en parallèle via le schéma d'Euler implicite (étape 4) . . . 142
21 Contrôle de l'ordonnanceur d'ATHAPASCAN . . . 143
22 Annotation de l'opérateur `Fork` avec des paramètres d'ordonnancement . 144
23 Spécification des politiques d'ordonnancement . . . 145
24 Algorithme du Gradient Conjugué pour la résolution de $Ax = b$ . . . 208

Liste des tableaux

3.1 Comparaison des méthodes d'intégration explicites [97] pour une pièce de tissu de période propre $T_0 \simeq \frac{1}{32}s$ 46

4.1 Tableau récapitulatif sur les méthodes d'intégration implantées (Euler explicite, Störmer-Verlet/leapfrog, Euler implicite) : stabilité, pas de temps, interactions des tâches de calculs, facilité de parallélisation. Les numéros de page relatifs aux explications étant indiqués entre parenthèses. 55

7.1 Les différents droits d'accès possibles pour les variables partagées `a1 : :Shared` 96

8.1 Comparaison de la complexité en communication dans les deux cas . . . 118

9.1 Structures de données de la simulation de textiles : type de données, nom des structures et termes raccourcis employés notamment dans les graphes de flots de données 122

10.1 Nombre de tâches créées durant la simulation parallèle : cas de l'emploi d'une méthode d'intégration explicite 151
10.2 Nombre de tâches créées durant la simulation parallèle : cas de l'emploi d'une méthode d'intégration d'Euler implicite 151
10.3 Tableau des temps d'exécution de chacune des tâches engendrées lors de la simulation avec l'emploi de la méthode d'intégration de Leapfrog . . . 154
10.4 Tableau des temps d'exécution de chacune des tâches engendrées lors de la simulation avec l'emploi de la méthode d'intégration d'Euler implicite 154
10.5 Temps T_1 et T_∞ de l'exécution dans le cas explicite 155
10.6 Temps T_1 et T_∞ de l'exécution dans le cas implicite 155
10.7 Mémoire utilisée dans chacune des structures de données de la simulation de textiles : cas de l'emploi de la méthode d'intégration d'Euler implicite 156
10.8 Mémoire utilisée dans chacune des structures de données de la simulation de textiles : cas de l'emploi d'une méthode d'intégration explicite 157

10.9 Nombre de tâches du GFD explicite placées sur 2 processeurs lors de l'emploi des stratégies Scotch (en haut) et Cyclic (en bas) pour 5 itérations de la simulation partitionnée en 4 sous-ensembles de particules 162
10.10 Nombre de tâches du GFD explicite placées sur 4 processeurs lors de l'emploi des stratégies Scotch et Cyclic pour 5 itérations de la simulation partitionnée en 4 sous-ensembles de particules 163
10.11 Nombre de tâches du GFD implicite placées sur 2 processeurs lors de l'emploi des stratégies Scotch (en haut) et Cyclic (en bas) pour 2 itérations de la simulation partitionnée en 4 sous-ensembles de particules 166
10.12 Nombre de tâches du GFD implicite placées sur 4 processeurs lors de l'emploi des stratégies Scotch et Cyclic pour 2 itérations de la simulation partitionnée en 4 sous-ensembles de particules 169

Introduction 1

Problématique

En informatique graphique, l'**animation de textiles** constitue un axe de recherche particulièrement important de par ses nombreuses applications dans le monde industriel auprès des industries de la confection et du tissage et grâce à son application directe dans les jeux vidéos ou encore dans les films d'animation, en permettant notamment l'habillage de personnages virtuels.

Dans le contexte industriel pour le développement d'activités liées aux textiles, la simulation doit impérativement être réaliste. C'est pourquoi elle est basée sur une **modélisation physique de l'objet déformable** afin de reproduire aux mieux son comportement. En effet afin d'obtenir un résultat réaliste, les lois fondamentales de la physique comme la vitesse, les forces (gravitation, vent, ...), les frottements, doivent être utilisées pour modéliser le mouvement de plusieurs objets interagissants. Les modèles employés pouvant alors être numériquement très complexes, le calcul d'une image de personnages varie de la seconde à plusieurs minutes suivant leur complexités.

L'une des difficultés principales réside alors dans l'obtention d'**animations interactives en temps réel**, c'est-à-dire dans la possibilité de calculer 25 images de la simulation de textiles par seconde en autorisant des interactions de la part de l'utilisateur. De telles simulations intéressent aussi bien les industries du textile que celles du loisir. Au niveau industriel, cette simulation intégrée aux logiciels de Conception Assistée par Ordinateur (CAO) permettrait de réduire considérablement les coûts de mise au point en ayant un aperçu direct de l'apparence d'un vêtement sans avoir à le retoucher. A un niveau plus ludique, il est facile d'imaginer les répercussions que cela aurait. Imaginez-vous dans un monde virtuel et voyez le réalisme que cela apporterait de se voir avec des vêtements

ayant un comportement semblable à la réalité avec l'intégration de tous les mouvements des tissus en réponse à vos propres mouvements ! Par contre, si nous souhaitons conserver la même complexité de simulation avec des modèles de tailles importantes, la diminution des temps de calcul nécessite obligatoirement la **parallélisation des algorithmes** employés.

Notre travail de recherche vise ainsi *l'obtention d'animations réalistes de textiles en temps réel, grâce à la parallélisation des algorithmes de simulation physique et leur exécution sur une grappe de machines*.

Afin de répondre à ce problème, ce travail de recherche, alliant calcul haute performance et informatique graphique, a fait l'objet d'une collaboration entre le laboratoire d'Informatique et Distribution (ID-IMAG) et le laboratoire GRAphisme, VIsion et Robotique (GRAVIR-IMAG). D'autre part ce travail a été partiellement financé par le contrat de la thématique prioritaire n°4 “Sciences et technologies de l'information, outils et applications” de la région Rhône-Alpes au travers du projet SAPPE, dans le cadre d'une collaboration avec la société Yxendis (Saint-Chamond), qui conçoit et commercialise des produits logiciels pour l'infographie textile, en vue, à terme, d'un transfert de technologie.

Le choix concernant l'utilisation d'une **grappe de machines** plutôt qu'une machine multiprocesseurs dédiée telle qu'un Cray est facilement compréhensible. En effet à l'heure actuelle, il est beaucoup moins coûteux de réunir plusieurs ordinateurs ordinaires que de se doter d'une machine unique multiprocesseurs. D'autre part, une grappe d'ordinateurs a l'avantage d'offrir une grande souplesse d'évolution grâce à la possibilité de rajouter à tout moment des processeurs. Cette dynamicité de l'architecture n'est hélas pas fournie par les machines multiprocesseurs. De plus de nos jours la grande majorité des entreprises sont dotées à la fois d'un réseau local et d'un parc informatique conséquent. Il est alors facile de maximiser les ressources matérielles existantes en employant par exemple ces machines la nuit en tant que grappe de calcul, alors que d'ordinaire elles sont inexploitées.

Afin d'exploiter au maximum les ressources offertes par une grappe de machines, nous avons utilisé l'**environnement de programmation parallèle ATHAPASCAN** [116, 104] développé au sein du laboratoire Informatique et Distribution (ID-IMAG) dans le cadre du projet APACHE-INRIA. Le parallélisme au sein de cet environnement est exprimé sous la forme de tâches de calculs s'exécutant en concurrence sur la machine parallèle cible. L'emploi de cet environnement, facilité par une interface applicative de haut niveau, permet d'assurer la portabilité des programmes créés en autorisant sans aucune modification des programmes une exécution aussi bien séquentielle, SMP (machines multiprocesseurs) que distribuée. Nous conservons ainsi au travers de la programmation, l'**extensibilité** offerte initialement par la grappe de machines.

Pour atteindre l'objectif de ce travail de recherche, plusieurs problèmes majeurs sont

à résoudre. Un premier point épineux concerne les calculs dynamiques, c'est-à-dire les calculs du mouvement du tissu au cours du temps. Un second point important concerne la détection de collisions. Elle permet notamment l'obtention des plis réalistes du tissu en contact avec une surface. Un troisième point majeur réside dans l'obtention des images issues de la parallélisation de la simulation. En effet, dans le domaine de l'informatique graphique, une simulation en temps réel n'a d'intérêt que si sa visualisation est possible.

Ce travail de recherche a ainsi débuté par l'implantation des calculs dynamiques, imposant des structures de données précises pour la parallélisation des algorithmes. Une phase d'analyse a également été effectuée pour l'intégration de la détection et du traitement des collisions au sein de l'application.

La visualisation de notre application est réalisée en utilisant l'**environnement de Réalité Virtuelle Net Juggler** [7, 5, 6, 99] développé initialement au sein du Laboratoire d'Informatique Fondamentale d'Orléans (LIFO) dont le développement se poursuit actuellement au sein du laboratoire ID-IMAG. Cet environnement permet notamment une visualisation de l'application sur plusieurs écrans en tirant partie des cartes graphiques disposées sur une grappe de machines standards.

L'emploi de cet environnement impose la réalisation d'un couplage entre une simulation parallèle et une visualisation également parallélisée. L'intérêt d'une telle combinaison peut permettre à terme une meilleure compréhension de la simulation de phénomènes très complexes nécessitant de gros volumes de données, dont la visualisation n'est généralement pas réalisable sans l'utilisation de machines fort coûteuses telles que les Silicon Graphics Incorporated (SGI).

Au niveau scientifique, l'étude d'algorithmes parallèles pour la synthèse d'animations implique un nouveau regard sur le domaine. En effet, la distribution des calculs sur une grappe de multiprocesseurs et les délais inévitablement variables de récupération des résultats remettent en cause l'approche synchrone utilisée jusqu'alors en synthèse d'animations. Un modèle asynchrone est ainsi nécessaire dans lequel les sous-ensembles (simulation parallèle, affichage, interaction) peuvent être exécutés indépendamment en l'absence d'actions synchronisées. Dans le domaine du parallélisme, ce projet pose également le problème des contraintes de temps réel et de leur implication sur l'architecture des systèmes. En effet, les applications généralement visées par le parallélisme nécessitent de longues heures voire des journées de calculs. Or dans le cadre de l'animation de textiles, les temps d'exécution sont de l'ordre de la minute, impliquant des tâches de calcul de petites tailles et nécessitant par conséquent un parallélisme à grain très fin.

Contributions

Les contributions apportées par ce travail de recherche sont les suivantes :

- **L'apport de nouvelles structures algorithmiques parallèles efficaces avec une approche parallèle encourageant le développement d'algorithmes asynchrones** permettant le déploiement d'applications de type serveurs d'animations capables de gérer efficacement un monde virtuel multi-utilisateurs. Cette parallélisation a fait l'objet de trois articles : [133, 134, 135].

- **La validation de l'approche de l'environnement de programmation parallèle ATHAPASCAN** avec notamment la mise au point d'applications ayant des contraintes temps réel mou, ainsi que le contrôle dynamique de son ordonnanceur. L'illustration de la facilité d'utilisation d'ATHAPASCAN via notamment l'application de simulation de textiles a fait l'objet d'un article : [102].

- **L'obtention d'un couplage entre une simulation parallélisée via** ATHAPASCAN **et sa visualisation effectuée sur plusieurs écrans grâce à l'emploi de l'environnement de Réalité Virtuelle Net Juggler**. Ce couplage de programmes parallèles a fait l'objet de deux articles : [136, 8].

Organisation du manuscrit

Ce travail de recherche s'intéresse à la parallélisation d'algorithmes de simulation physique dédiée à la synthèse d'images avec un intérêt plus marqué concernant l'animation de textiles. Ce document est composé de cette introduction et de quatre parties comportant un total de onze chapitres. Le manuscrit se termine par une galerie d'images issues de la simulation de textiles.

Avant de vouloir aborder la création d'algorithmes parallèles destinés aux simulations utilisées en informatique graphique, il est primordial de comprendre leurs particularités. La première partie du document est ainsi consacrée à la présentation de **la simulation de textiles**. Quels sont les objectifs d'une telle simulation ? Comment réussir à *modéliser un bout d'étoffe* et à lui faire *simuler un comportement réaliste* ? Les réponses à ces questions sont l'objet du second chapitre.

Quelques outils mathématiques sont nécessaires à cette simulation. Dans notre cas, l'animation de textiles passe par l'emploi de *méthodes d'intégration numérique*. Le chapitre trois en détaillera trois sur lesquelles notre choix s'est porté.

Puis le chapitre quatre établit un bref résumé de l'ensemble des calculs que comporte la simulation en mettant en évidence *les sources de parallélisme* de cette application.

Après cette analyse sur l'objet de notre simulation, nous fournissons quelques éléments de base indispensables à la compréhension future de notre propre parallélisation. C'est pourquoi la seconde partie du document fournit une présentation théorique du **parallélisme** en vue de son application dans le cadre à la fois de l'animation de textiles et de l'emploi d'une grappe comme machine parallèle. Le chapitre cinq intitulé *calcul parallèle*, explique les différentes étapes de la conception d'un algorithme parallèle avec comme point de départ les spécifications du problème initial. Quelques critères de performances sont également répertoriés permettant d'évaluer la qualité de l'algorithme parallèle créé.

La parallélisation de notre application de textiles a été élaborée en employant l'environnement de programmation parallèle ATHAPASCAN avec une expérimentation effectuée sur une grappe de machines. Quels sont les intérêts liés à l'utilisation de cet environnement ? Pourquoi employer ce type de *machine parallèle* et non pas une machine multiprocesseurs de type Cray ? Pourquoi ne pas avoir utilisé un *environnement de programmation parallèle* de type échange de messages (MPI) ? Les réponses à ces questions sont fournies au sein du chapitre six.

La chapitre sept présente d'une part l'interface de programmation de l'environnement **ATHAPASCAN** et d'autre part explique comment cet environnement prend en charge les tâches de calcul, interprète leurs dépendances de données afin de les exécuter dans le respect de ces contraintes sur la grappe de machines.

La troisième partie du manuscrit est consacrée essentiellement à notre travail de recherche sur **la parallélisation et le couplage de programmes parallèles** dans le cadre de l'animation de textiles. Le chapitre huit présente le *partitionnement* qui a été utilisé pour découper l'objet déformable, sujet de la simulation, en morceaux sur lesquels les tâches de calculs seront exécutées.

Le chapitre neuf détaille l'ensemble des *tâches de calculs créées* au sein de la simulation. De plus, il ne faut pas oublier que le choix de l'*ordonnancement de ces tâches* a un rôle important dans l'obtention de bonnes performances. C'est pourquoi ce chapitre montre également la facilité que procure ATHAPASCAN pour tester différents ordonnancements des tâches de l'application.

Le chapitre dix présente l'ensemble des résultats de notre parallélisation en terme de *performances*. Quelles sont les complexités théoriques et pratiques de nos algorithmes ? Quel est le surcoût dû à l'emploi de l'environnement ATHAPASCAN ? Comment choisir la granularité optimale pour un problème donné ? Les communications sont elles recouvertes par les calculs ? Voici quelques exemples des questions auxquelles nous tenterons de répondre dans ce chapitre.

En informatique graphique une simulation de textiles n'a pas un grand intérêt si l'objet simulé ne peut être visualisé. C'est pourquoi il est nécessaire d'effectuer un *couplage entre la simulation parallèle et la visualisation*, elle-même parallélisée pour être visionnée sur plusieurs écrans. Après une présentation de l'emploi de plus en plus fréquent de

grappe de machines en Réalité Virtuelle, le chapitre onze détaille le couplage que nous avons élaboré.

Enfin le chapitre douze conclut ce mémoire de thèse par une *analyse des contributions* apportées par notre travail et par une discussion sur les *travaux futurs*.

Pour compléter ce mémoire, la quatrième partie comporte l'ensemble des **annexes**. L'annexe A est consacrée à la *méthode du Gradient Conjugué* intervenant dans l'application d'un schéma d'intégration implicite pour la résolution du système d'Équations Différentielles Ordinaires modélisant le comportement du tissu. En effet, il n'est pas forcément nécessaire d'avoir une compréhension totale de cette méthode lors de son utilisation, mais il semble cependant important d'avoir la possibilité d'approfondir cette connaissance par la suite. Pour finir ce manuscrit, l'annexe B présente une *galerie d'images* issues de la simulation de textiles.

La simulation de textiles

L'un des objectifs de ce travail de recherche est l'obtention d'algorithmes parallèles de simulations physiques d'objets déformables employées en synthèse d'images. Afin de mieux comprendre les caractéristiques spécifiques à ce type d'animation, nous avons fait cette étude dans le cadre particulier de l'animation de textiles par modèles physiques.

Cette partie présente donc les modèles physiques les plus employés pour simuler le mouvement d'un tissu. Nous verrons qu'ils peuvent être plus ou moins complexes selon l'objectif recherché : reproduire un mouvement qui semble réel ou bien reproduire le mouvement exact prenant en compte toutes les propriétés physiques du textile considéré. Nous verrons que la formulation de l'équation du mouvement de Newton conduit à un système différentiel que nous devons résoudre pour obtenir l'état de notre tissu. Nous présenterons alors plusieurs méthodes d'intégration numérique qui diffèrent par leur degré de stabilité et par leur complexité.

Modèles physiques 2

Dans le cadre de cette thèse, nous nous intéressons essentiellement à la simulation de textiles par modèles physiques. A travers ce chapitre, nous allons donc étudier les différents modèles physiques usuellement employés pour la simulation de textiles.

2.1 Introduction

La modélisation de textiles peut être abordée de plusieurs manières selon le but recherché. Lors de la création d'images de synthèses pour les jeux vidéos, l'objectif est de produire des images réalistes et convaincantes. Dans ce cas, les contraintes physiques sont ignorées ou du moins considérablement simplifiées. En effet, les infographistes préfèrent souvent simplifier le calcul d'une image en simulant “à la main” un mouvement plausible plutôt que de calculer le mouvement réel. En informatique graphique, les recherches se sont plutôt portées sur des simulations basées sur des modèles physiques permettant de restituer un comportement approché du tissu. Les travaux ne tendent pas à reproduire le mouvement exact du textile en fonction de sa matière. Dans d'autres cas, le but est au contraire de préserver les propriétés physiques du textile afin de simuler son comportement réel. Prenons l'exemple de logiciels de CAO pour les industries du textile. Leur objectif est de reproduire le comportement exact d'un tissu selon sa matière. Cela signifie que le tissu doit être modélisé de façon à prendre en compte ses propriétés physiques afin de réussir à simuler ses déformations et ses mouvements au cours du temps.

Dans le cadre de cette thèse, nous nous intéressons essentiellement à la simulation de textiles par modèles physiques utilisée dans le domaine de l'informatique graphique [80, 125, 89, 30]. Au travers de ce chapitre, nous allons étudier les différents modèles phy-

siques [131, 124, 81] usuellement employés pour la simulation de textiles. Il faut savoir qu'il existe essentiellement deux types de modélisation. La première utilise des modèles *discrets* tandis que la seconde utilise des modèles *continus*. Nous étudierons ces deux types de modélisation avec une préférence pour la première qui semble plus facile à implanter.

En informatique graphique, la validation des modèles et des algorithmes employés lors de simulations est souvent effectuée de manière visuelle. Par exemple, pour la simulation de textiles, les articles sont généralement accompagnés de vidéos montrant le comportement réaliste du tissu simulé. De plus, les modèles employés dans ces simulations comportent généralement un certain nombre de paramètres devant être initialisés de façon adéquate afin de reproduire le comportement souhaité. Cette phase est primordiale et souvent délicate à réaliser. En effet, un modèle réputé réaliste peut engendrer un comportement instable de l'objet si les valeurs des paramètres ont été mal ajustées. C'est pourquoi nous pouvons noter que cette année, Bhat et *al.* [20] ont publié un article présentant une méthode permettant d'estimer ces paramètres basée sur une capture vidéo de l'objet réel.

Par ailleurs, nous pouvons citer un livre dédié à l'animation et la modélisation de textiles [64] écrit par Donald House et David Breen en 2000, regroupant l'ensemble des articles faisant date dans l'animation de vêtements et qui présente également un survol des techniques employées dans le monde de l'industrie du textile.

2.2 Modèles discrets

Les modèles physiques que nous allons présenter au sein de ce chapitre se basent sur des réseaux de particules en interaction appelés *systèmes de particules*.

Les systèmes de particules ont été présentés au sein de la communauté d'informatique graphique en 1983 par Reeves [100]. Reeves décrivait alors une technique utilisée pour simuler du feu balayant une planète dans la Séquence Génésis du film *Star Trek II : The Wrath of Khan*. Il définit alors le modèle de système de particules comme "un nuage de particules primitives", dans lequel chaque particule est générée au sein du système, bouge, interagit avec les autres, change de forme et de couleur et enfin meurt. Reeves et Blau [101] présentent ensuite le concept de systèmes de particules structurés pour créer des arbres et de l'herbe. Les systèmes de particules sont de plus en plus utilisés pour simuler un grand nombre de phénomènes physiques, en particulier les textiles [25, 40, 39].

Les modèles dynamiques que nous allons aborder se basent donc sur une discrétisation du vêtement en un maillage polygonal. Les sommets du maillage sont appelés particules ou masses [118, 25]. La topologie du maillage définit les interactions et les forces exer-

cées entre les particules. Nous obtenons alors une représentation dynamique paramétrée de l'objet respectant au mieux sa forme géométrique. La figure 2.1 illustre le cas de la discrétisation d'un textile en un maillage polygonal.

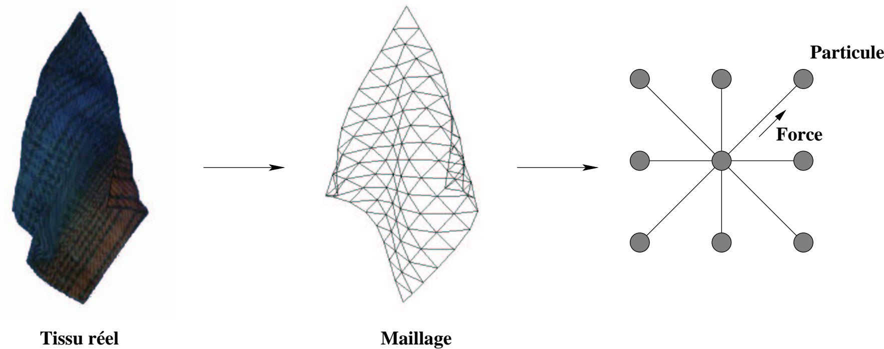


Figure 2.1 *Discrétisation du textile en un maillage polygonal. Les sommets sont appelés particules. Sa topologie définit les interactions et les forces entre les particules.*

Les modèles se différencient seulement entre eux par la définition des interactions qui existent entre les particules, c'est-à-dire qu'il y a une phase de calibration de la représentation afin qu'elle respecte au mieux les propriétés dynamiques de l'objet.

Nous avons à notre disposition le maillage décrivant la topologie du vêtement, c'est-à-dire que nous connaissons le *voisinage* de chacune des particules constituant le tissu. A partir de cette information, il est possible de calculer les forces exercées sur chacune des particules. Celles-ci sont calculées à partir de la position et de la vitesse de la particule considérée, ainsi que des position et vitesse de ses voisines. Nous pouvons dès lors noter que cette notion de *voisinage* est primordiale pour ce calcul et donc pour effectuer une simulation de textiles.

A partir du moment où nous avons réussi à déterminer la fonction f calculant les forces, le mouvement des particules est gouverné par l'équation du mouvement de Newton. La trajectoire de chacune des particules ayant une masse m_i et une position x_i est calculée à partir de l'équation :

$$f_i(x, v) \quad = \quad m_i \frac{d^2 x_i}{dt^2}. \tag{2.1}$$

Le vecteur x représente le vecteur contenant toutes les positions des particules, et le vecteur v celui des vitesses des particules. Les différences qui existent entre les systèmes résident dans les méthodes employées pour calculer les forces.

2.2.1 Systèmes masses-ressorts de Provot

Dans les systèmes dits “masses-ressorts” [38], les interactions entre les particules sont modélisées à l'aide de ressorts généralement linéaires. Provot [96, 97] propose un système masses-ressorts pour la modélisation des textiles. Il utilise un maillage rectangulaire dans lequel les particules sont connectées par différents ressorts. Sur la figure 2.2 nous pouvons observer les différents ressorts introduits dans le modèle de Provot : les ressorts longitudinaux (1) permettent de contrebalancer les tensions, les ressorts diagonaux (2) s'opposent au cisaillement, et les ressorts courbés s'opposent aux flexions (3).

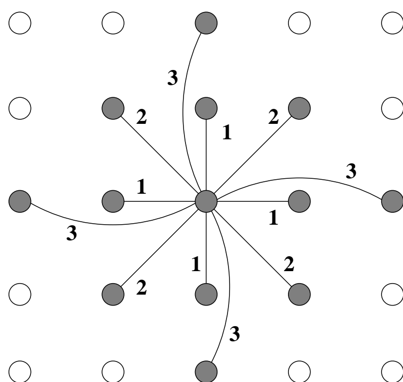


Figure 2.2 *Le système masses-ressorts de Provot avec (1) les ressorts longitudinaux, (2) les ressorts diagonaux et (3) les ressorts de flexion*

Le système masses-ressorts proposé par Provot ne permet pas de modéliser des comportements spécifiques à un matériau donné ne s'appuyant pas sur les propriétés des tissus réels. Mais il est capable de reproduire visuellement des animations qui sont suffisantes dans la majorité des applications d'informatique graphique.

La force élastique due au ressort linéaire reliant deux particules i et j, de positions x_i et x_j est donnée par :

$$\left\{ \begin{array}{lcl} f^e_{i,j}(x) & = & k_{i,j}\left(\|x_i - x_j\| - l_{ij}\right)\frac{x_i - x_j}{\|x_i - x_j\|}, \\ f^e_{i,j}(x) & = & f^e_{j,i}(x), \end{array} \right.$$

où k_{ij} est la raideur du ressort considéré, et l_{ij} sa longueur au repos. La constante de raideur dépend du type de ressort. La valeur des constantes est importante pour les ressorts de construction, par contre pour les ressorts de flexion et de tension les valeurs sont petites.

Afin de tenir compte de l'énergie dissipée à cause de frictions internes, nous devons considérer également les forces de viscosité. Ces forces permettent d'amortir l'énergie

cinétique et dépendent de la vitesse de l'objet. Elles sont généralement modélisées pour chaque ressort par la fonction suivante :

$$f_{i,j}^{v}(x) = \nu_{i,j}(v_i - v_j), \tag{2.2}$$

où ν_{ij} est le coefficient de viscosité du ressort reliant la particule i à la particule j. Ce terme étant linaire, il est particulièrement bien adapté pour une intégration numérique. Mais il pénalise également une rotation stricte d'un ressort et une viscosité importante empêche les flexions de l'objet. De plus, le fait d'exprimer ainsi la viscosité rend le mouvement d'un objet déformable assez raide. Pour remédier à ces inconvénients, la viscosité du ressort peut être modélisée par :

$$f_{i,j}^{v}(x) = \nu_{i,j}\frac{< v_i - v_j, x_i - x_j >}{\|x_i - x_j\|^2}(x_i - x_j), \tag{2.3}$$

Le terme linaire (2.2) a simplement été projeté dans la direction du ressort. Hélas, dans la majorité des cas, ce terme va compliquer l'intégration implicite que nous étudierons un peu plus loin.

En résumé, nous avons à sommer et à intégrer les forces des ressorts dans l'équation du mouvement (2.1) pour effectuer la simulation. Dans l'implantation de notre simulation de textiles, nous avons utilisé une modélisation de type masses-ressorts comportant seulement les ressorts (1) et (2) du modèle de Provot, c'est-à-dire comportant des ressorts longitudinaux et diagonaux mais sans ressorts de flexions. Nous avons en effet préféré partir d'un modèle plus simple, mais équivalent en terme de parallélisme. Par la suite, les ressorts de flexion pourront être rajoutés à notre modèle sans difficulté majeure, ne modifiant en rien la méthode de parallélisation. En effet, l'ajout de ces ressorts élargira simplement le voisinage de chaque particule, mais en gardant une notion de voisinage locale à une particule donnée.

Limitation de la déformation

Afin de simuler le comportement rigide des tissus, Provot [97] introduit une notion de limitation des déformations élastiques des ressorts. En effet, il fait remarquer à juste titre dans sa thèse que les tissus se déforment peu en terme de traction, compression ou cisaillement mais que par contre ils se déforment énormément en terme de flexion. C'est pourquoi il soulève l'hypothèse que la modélisation élastique n'est peut être pas la plus adaptée pour rendre compte de tous ces comportements.

Provot a ainsi opté pour une méthode basée sur le principe de limitation des déformations élastiques des ressorts par l'ajout d'une contrainte du type $\tau \leq \tau_{max}$ où τ est le *taux d'élongation* de chaque ressort et τ_{max} le taux d'élongation *maximum* que le ressort ne doit pas dépasser. En pratique ce taux est fixé à 10%. Le taux d'élongation d'un ressort étant défini par $\tau = \frac{l-l_0}{l_0}$ où l est la longueur du ressort et l_0 sa longueur à vide. Chaque

ressort peut ainsi se déformer élastiquement jusqu'à un certain taux d'élongation maximum τ_{max}, et au-delà il ne peut plus du tout se déformer, devenant une tige indéformable. Le modèle de Provot prend ainsi en compte les deux régimes mécaniques de la déformation des tissus : le régime de faible raideur et celui de forte raideur dans lequel il n'y a aucune déformation.

2.2.2 Modèle de Kawabata

Nous pouvons noter qu'il existe des modélisations qui prennent en compte les propriétés spécifiques d'un tissu donné en utilisant notamment les mesures faites par Kawabata [72]. Ces séries de mesures mettent en évidence les comportements mécaniques de différentes étoffes en suivant cinq types de sollicitations : traction, cisaillement, flexion, compression et friction. La figure 2.3 donne l'exemple d'une courbe représentant la flexion d'un tissu 100% coton.

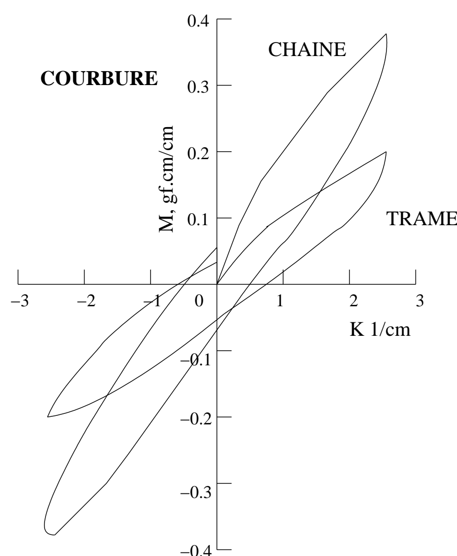


Figure 2.3 *Courbe de Kawabata présentant la flexion d'un tissu 100% coton*

Pour cela, le maillage rectangulaire décrit par les masses et les ressorts de construction du système masses-ressorts proposé par Provot peut être considéré comme un réseau de fils tissés, dans lequel un fil est donné par une chaîne de ressorts. Des forces plus complexes que celles vues précédemment pour les ressorts linéaires peuvent être rajoutées au modèle afin de modéliser les interactions entre les fils. Un système à particules défini de façon plus général est ainsi obtenu.

Dans la plupart des systèmes à particules, des fonctions sont utilisées pour décrire les énergies du système dues aux pressions, tensions, et flexions exercées sur l'objet déformable. Ces énergies sont choisies de façon à ce qu'elles correspondent aux mesures

expérimentales faites par Kawabata afin de reproduire les propriétés propres au textile considéré. Toutes ces énergies sont modélisées sur une grille rectangulaire où chaque particule interagit avec ses quatre voisins directs. La grille est alignée selon deux directions distinctes qui sont apparentes dans les textiles (pour les textiles tissés elles sont appelées trame et chaîne). En effet, ces deux directions permettent de mettre en évidence différentes propriétés du tissu, c'est pourquoi chaque expérimentation doit être faite selon ces deux directions. La force $\vec{f}$ est alors calculée à partir des énergies engendrées par le maillage [25].

La différence entre cette représentation et celle utilisée par Provot réside essentiellement dans le calcul des forces appliquées sur chacune des particules. Dans le modèle de Provot, les particules interagissent les unes sur les autres via différents types de ressorts. Les forces appliquées sur les particules correspondent donc aux forces exercées par ces ressorts. Par contre dans la représentation issue du système d'évaluation de Kawabata, le maillage est tout d'abord considéré comme étant rectangulaire, et ensuite les forces exercées sur les particules sont calculées à partir des énergies produites par les quatre voisins de chacune des particules. Le calcul des énergies comporte par ailleurs des paramètres matériaux non présents dans le modèle de Provot, permettant de tenir compte des propriétés spécifiques à un type de textile donné.

2.2.3 Formulation de l'équation du mouvement

La modélisation de la simulation de vêtements que nous avons implantée est basée sur un maillage triangulaire de N particules dans l'espace $\mathbb{R}^3$. Sur chacune de ces particules est exercée une force $\vec{f_i}$. Pour conserver les propriétés d'élasticité et de poids du tissu, une représentation mécanique avec un système de type masses-ressorts est utilisée : les ressorts sont positionnés entre les particules pour permettre d'exercer des forces sur ces masses et ainsi fournir un comportement réaliste au tissu.

La simulation consiste à calculer à chaque pas de temps, les états des particules (position, vitesse, accélération). Pour cela, nous allons voir qu'au préalable il est nécessaire de calculer les forces appliquées sur chacune des particules. Ensuite nous verrons que le système de particules est décrit par un système différentiel que nous aurons à résoudre pour obtenir les états des particules.

Forces exercées sur les particules

L'accélération de la $i^{ème}$ particule de la simulation dynamique est donnée en appliquant la loi fondamentale de la dynamique énoncée par Newton, soit

$$m_i x_i''(t) = \vec{f_i}(t), \tag{2.4}$$

où $\vec{f}_i(t)$ est la force exercée sur cette particule au temps t et m_i sa masse.

Les forces appliquées au temps t sur chacune des particules, illustrées par la figure 2.4, sont dues aux forces exercées par les ressorts (forces locales), mais également à des forces extérieures telles que la gravité ou encore le vent (forces globales).

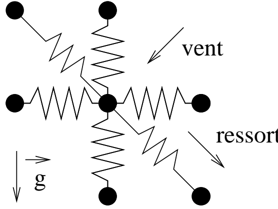


Figure 2.4 *Forces appliquées sur une particule : forces dues aux ressorts (forces locales) et forces extérieures (gravité, vent : forces globales)*

La force appliquée à la $i^{ème}$ particule du système peut ainsi être formulée :

$$\vec{f}_i(t) = \sum_{j|(i,j)\in E} \left[\vec{f}^e_{i,j}(t) + \vec{f}^v_{i,j}(t)\right] + m_i\vec{g} + \vec{f}_{externe}(t), \tag{2.5}$$

avec :

$$\begin{cases} \vec{f}^e_{i,j}(t) &= k_{ij}\left(\|x_i(t) - x_j(t)\| - l_{ij}\right)\ \vec{u}_{ij}(t), \\ \vec{f}^v_{i,j}(t) &= \nu_{ij}\left(v_i(t) - v_j(t)\right)\ \vec{u}_{ij}(t)\ \vec{u}_{ij}(t), \end{cases}$$

où :

- E représente l'ensemble des arêtes du système à particules,
- k_{ij} est la constante de raideur des ressorts,
- ν_{ij} est la constante d'amortissement des ressorts,
- l_{ij} est la longueur du ressort au repos reliant la particule i et la particule j,
- $\vec{g}$ représente le vecteur de la gravité,
- et $\vec{u}_{ij}(t)$ est un vecteur unitaire défini par $\vec{u}_{ij}(t) = \frac{x_i(t)-x_j(t)}{||x_i(t)-x_j(t)||}$.

Système différentiel associé au système de particules

Définissons tout d'abord les différents vecteurs nécessaires à la simulation de notre système de N particules. Ces vecteurs sont tous de taille $3N$, chacune de leurs composantes étant définies suivant les 3 directions x, y et z. Le vecteur position x de toutes les particules est défini par

$$x(t) = \{x_1(t), ..., x_N(t)\},$$

le vecteur x' des vitesses des particules par

$$x'(t) = \{x'_1(t), ..., x'_N(t)\}$$

et le vecteur x'' des accélérations des particules par

$$x''(t) = \{x''_1(t), ..., x''_N(t)\}.$$

Le vecteur f est quant à lui celui des forces appliquées sur chacune des particules, défini par

$$f(t) = \{f_1(t), ..., f_N(t)\}.$$

Soient $m_1, ..., m_N$ les masses des N particules contenues dans le système, nous pouvons définir une matrice diagonale M de taille $3N \times 3N$ telle que $M = diag\,(m'_1, ..., m'_N)$, avec

$$m'_i = \begin{pmatrix} m_i & 0 & 0 \\ 0 & m_i & 0 \\ 0 & 0 & m_i \end{pmatrix}.$$

Le système mécanique peut alors être exprimé sous la forme d'un système d'Équations Différentielles Ordinaires (EDO) du second ordre accompagné de valeurs initiales :

$$\begin{cases} x''(t) &= M^{-1} f\,(t, x(t), x'(t))\,, \\ x'(t_0) &= v_0, \\ x(t_0) &= x_0. \end{cases} \tag{2.6}$$

De plus, si une variable séparée caractérisant les vitesses est introduite, définie par

$$v(t) = x'(t),$$

ce système d'équations différentielles peut être transformé en un système du premier ordre dans lequel les dérivées sont établies par rapport au temps t :

$$\begin{pmatrix} v(t) \\ x(t) \end{pmatrix}' = \begin{pmatrix} x'(t) \\ x(t) \end{pmatrix}' = \begin{pmatrix} M^{-1} f\,(t, x(t), v(t)) \\ v(t) \end{pmatrix}, \begin{pmatrix} v(t_0) \\ x(t_0) \end{pmatrix} = \begin{pmatrix} v_0 \\ x_0 \end{pmatrix}. \tag{2.7}$$

La résolution de ce système permet l'obtention des vitesses et positions des particules. Cette résolution s'effectue en employant un schéma d'intégration. Trois schémas seront présentés en détail dans le chapitre suivant, dont un dit implicite. Dans la section 3.5.3, nous verrons que l'emploi de ce schéma d'intégration implicite va nécessiter le calcul de deux matrices particulières, $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial v}$, appelées matrices des dérivées des forces.

Matrices des dérivées des forces : Volino et Magnenat-Thalmann

Les matrices $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial v}$ représentent les variations des forces d'élasticité et de viscosité par rapport à la position et à la vitesse des particules. Volino et Magnenat-Thalmann introduisent la notion de matrices dites de contribution des forces [127] qui mettent en évidence les forces élémentaires d'interaction exercées entre les particules. Ces matrices des dérivées partielles $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial v}$ sont des matrices creuses de taille $3N \times 3N$ avec N le

nombre de particules contenues dans le système, chacun de leurs éléments étant de taille 3×3.

Dans un système masses-ressorts, les interactions correspondent aux interactions qui existent entre les couples de particules. Les relations de force sont définies par les forces d'élasticité linéaire et les forces d'amortissement

$$f_{ij} = k_{ij}(l_{ij} - \|x_i - x_j\|)\vec{u_{ij}} - \nu_{ij}(v_i - v_j)\vec{u_{ij}}\vec{u_{ij}}$$

avec $\vec{u_{ij}} = \frac{x_i - x_j}{\|x_i - x_j\|}$ l'orientation du ressort, l_{ij} la longueur du ressort reliant les particules i et j à l'initialisation, k_{ij} la constante de raideur du ressort et ν_{ij} sa constante d'amortissement. Les contributions des dérivées des forces d'élasticité et de viscosité sont alors définies par

$$\left\{ \begin{array}{l} \frac{\partial f_i}{\partial x_j} = k_{ij} u_{ij} u_{ij}^T \\ \frac{\partial f_i}{\partial x_i} = \sum_j \frac{\partial f_i}{\partial x_j} \end{array} \right. \quad et \quad \left\{ \begin{array}{l} \frac{\partial f_i}{\partial v_j} = \nu_{ij} u_{ij} u_{ij}^T \\ \frac{\partial f_i}{\partial v_i} = \sum_j \frac{\partial f_i}{\partial v_j} \end{array} \right. . \qquad (2.8)$$

Dans le cas général, caractérisé par une modélisation basée sur un maillage irrégulier, les matrices $\frac{df}{dx}$ et $\frac{df}{dv}$ associées au maillage sont diagonales et creuses. La figure 2.5 présente la forme de ces matrices. Sur la ligne i de la matrice, correspondant à la particule i, les éléments non nuls en dehors de l'élément diagonal correspondent aux voisins de cette particule. En effet, les éléments non nuls hors diagonale de ces matrices correspondent aux $\frac{\partial f_i}{\partial x_j}$ et $\frac{\partial f_i}{\partial v_j}$ définis pour les paires de particule i et j reliées entre elles. L'élément diagonal représente la somme des contributions des forces dues à l'ensemble des voisins de i, alors que l'élément de la colonne j de la ligne i correspond seulement à la contribution du voisin j de la particule i.

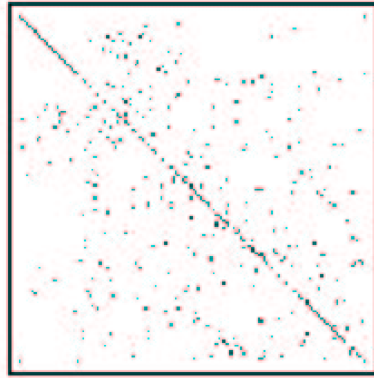


Figure 2.5 *Forme des matrice des dérivées des forces $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial v}$. Les éléments diagonaux sont non nuls. Ils correspondent à la somme des contributions des forces de tous leurs voisins. Les éléments non nuls en dehors de la diagonale représentent leurs propres contributions par rapport à la particule de la diagonale.*

Dans l'implantation du schéma implicite entraînant le calcul de ces matrices des dérivées des forces, nous avons utilisé les interactions définies par Volino et Magnenat-Thalmann, mais il existe d'autres définitions.

Matrices des dérivées des forces : Baraff et Witkin

Des interactions plus complexes peuvent être définies de façon similaire. Baraff et Witkin [13] introduisent des fonctions de contraintes afin de calculer les forces et les dérivées des forces. En effet, la position et la vitesse d'une particule donnée peuvent être complètement contrôlées dans une, deux ou même dans les trois dimensions. Les particules peuvent ainsi être attachées à un point fixe ou mobile dans l'espace. Les contraintes peuvent être soit définies par l'utilisateur, soit automatiquement générées suite à un contact entre le tissu et un solide par exemple. C'est pourquoi Baraff et Witkin ont choisi de définir le comportement interne du tissu en formulant un vecteur de contraintes $C(x)$ qui atteint zéro quand la configuration souhaitée est obtenue.

Par exemple, si nous souhaitons que deux particules a et b atteignent la même position, nous définirons une fonction de contrainte $C(x)$ telle que $C(a, b) = a - b$. Si par contre nous souhaitons que a et b soient distants d'une longueur r, la fonction de comportement adéquate serait alors $C(a, b) = |a - b| - r$.

A ce vecteur de comportement ou de contraintes est associée une énergie E_c définie par

$$E_c(x) = \frac{k}{2} C(x)^T C(x)$$

où k représente la constante de raideur des ressorts. En supposant que C ne dépende que d'un faible nombre de particules, C donne lieu à un vecteur de force creux f, où chaque élément f_i est un vecteur dans $\mathbb{R}^3$. Pour chaque particule i pour laquelle C a une dépendance,

$$f_i = -\frac{\partial C(x)}{\partial x_i} \left(kC(x) + \nu C'(x) \right), \tag{2.9}$$

avec ν la constante d'amortissement des ressorts et $C'(x)$ la dérivée par rapport au temps t. Les autres éléments de f sont nuls. La dérivée $C'(x)$ est approchée par

$$C'(x) = \frac{\partial C(x)}{\partial t} = \left(\frac{\partial C(x)}{\partial x} \right)^T \frac{\partial x}{\partial t} = \left(\frac{\partial C(x)}{\partial x} \right)^T v. \tag{2.10}$$

De même que f, la dérivée de f est creuse. Soit la matrice des dérivées partielles K définie telle que $K = \frac{\partial f}{\partial x}$. Les éléments non nuls de K correspondent aux K_{ij} définis pour les paires de particule i et j pour lesquelles C a des dépendances. La matrice K est de taille $3N \times 3N$ avec N le nombre de particules contenues dans le système, chacun de ses

éléments K_{ij} étant de taille 3 × 3. A partir de l'équation (2.9), nous en déduisons

$$K_{ij} = \frac{\partial f_i}{\partial x_j} = -k \frac{\partial C(x)}{\partial x_i} \frac{\partial C(x)^T}{\partial x_j} - \frac{\partial^2 C(x)}{\partial x_i \partial x_j} \left(kC(x) + \nu C'(x) \right). \tag{2.11}$$

Nous pouvons noter que la matrice K est symétrique car $K_{ij} = K_{ji}^T$ et que si C ne dépend pas de v, la matrice $\frac{\partial f}{\partial v}$ est alors nulle (figure 2.5). La dérivée $\frac{\partial f_i}{\partial v_j}$ est approchée par

$$\frac{\partial f_i}{\partial v_j} = -\nu \frac{\partial C(x)}{\partial x_i} \frac{\partial C'(x)^T}{\partial v_j} = -\nu \frac{\partial C(x)}{\partial x_i} \frac{\partial C(x)^T}{\partial x_j}. \tag{2.12}$$

Il ne reste plus qu'à définir les fonctions de contraintes afin de décrire les forces internes agissant sur le textile.

Forces d'élasticité

Chaque particule possède une position x_i dans l'espace 3D qui change au cours du temps et des coordonnées 2D (u_i, v_i) représentant la position de la particule quand le tissu est représenté sous la forme d'une surface planaire. Ces coordonnées ne sont quant à elles pas modifiées au cours de la simulation. Malgré le fait que le tissu soit modélisé sous la forme d'un ensemble discret de points regroupés en triangle, il est pour le moment plus adéquat de prétendre qu'il existe une fonction continue $w(u, v)$ permettant de passer des coordonnées planaires aux coordonnées spatiales. L'élasticité peut être mesurée en chaque point de la surface du tissu en examinant les dérivées $w_u = \frac{\partial w}{\partial u}$ et $w_v = \frac{\partial w}{\partial v}$ en ce point. La magnitude de w_u décrit l'élongation ou la compression dans la direction u. Le tissu est inextensible quand $\|w_u\| = 1$. L'élasticité dans la direction v est mesurée par $\|w_v\|$.

Nous appliquons cette mesure d'élongation/compression à un triangle. Pour cela, considérons un triangle dont les sommets sont les particules i, j et k. Soient

$$\begin{cases} \Delta x_1 = x_j - x_i \\ \Delta x_2 = x_k - x_i \end{cases}, \quad \begin{cases} \Delta u_1 = u_j - u_i \\ \Delta u_2 = u_k - u_i \end{cases}, \quad \begin{cases} \Delta v_1 = v_j - v_i \\ \Delta v_2 = v_k - v_i \end{cases}. \tag{2.13}$$

Nous approximons $w(u, v)$ par une fonction linéaire sur chaque triangle, ce qui équivaut à dire que w_u et w_v sont constants sur chaque triangle. Ceci nous permet d'écrire

$$\begin{cases} \Delta x_1 = w_u \Delta u_1 + w_v \Delta v_1 \\ \Delta x_2 = w_u \Delta u_2 + w_v \Delta v_2 \end{cases} \tag{2.14}$$

que nous pouvons écrire sous la forme d'un système permettant la recherche de w_u et w_v

$$\begin{pmatrix} w_u & w_v \end{pmatrix} = \begin{pmatrix} \Delta x_1 & \Delta x_2 \end{pmatrix} \begin{pmatrix} \Delta u_1 & \Delta u_2 \\ \Delta v_1 & \Delta v_2 \end{pmatrix}^{-1} \tag{2.15}$$

ou encore

$$\begin{pmatrix} w_u & w_v \end{pmatrix} = \begin{pmatrix} \Delta x_1 & \Delta x_2 \end{pmatrix} \begin{pmatrix} \frac{\Delta v_2}{det} & -\frac{\Delta u_2}{det} \\ -\frac{\Delta v_1}{det} & \frac{\Delta u_1}{det} \end{pmatrix}, \tag{2.16}$$

avec det le déterminant de la matrice défini en utilisant la règle de Cramer, soit

$$det = \Delta u_1 \Delta v_2 - \Delta v_1 \Delta u_2.$$

Nous pouvons traiter w_u et w_v comme des fonctions dépendantes de x en ayant conscience qu'elles ne dépendent en fait que de x_i, x_j et x_k. L'équation (2.15) est utilisée pour calculer les dérivées de $w(u, v)$ par rapport aux positions x_i, x_j, et x_k des particules i, j, et k. La fonction de contrainte que nous utilisons pour l'énergie d'élasticité est alors

$$C(x) = a \begin{pmatrix} \|w_u(x)\| - b_u \\ \|w_v(x)\| - b_v \end{pmatrix}, \tag{2.17}$$

avec a l'aire du triangle exprimée selon les coordonnées uv. Cette fonction est nulle quand le triangle donné par les particules i, j et k est inélastique. En général nous posons $b_u = b_v = 1$, mais elles peuvent être augmentées ou diminuées selon les parties du tissu considérées. En particulier, si nous souhaitons légèrement allonger un vêtement (une chemise par exemple) dans la direction u, nous pouvons alors augmenter b_u, impliquant une valeur plus importante pour w_u et ainsi provoquer des plis le long de la direction u. De la même manière, nous pouvons diminuer b_v en bas de la manche et ainsi simuler des manchettes raides comme pour un sweat-shirt. Cette fonction permet donc de contrôler l'élasticité du tissu.

Forces de tension

Par ailleurs, un tissu résiste à un cisaillement dans le plan. Nous pouvons mesurer l'allongement dans un triangle, en considérant le produit scalaire $w_u^T w_v$. Dans l'état d'équilibre, le produit est nul. Comme le terme d'élasticité influe sur la magnitude de w_u et w_v en évitant les changements brutaux, il n'est pas nécessaire de normaliser. Le produit $w_u^T w_v$ est une approximation de l'angle de tension. La contrainte de tension peut donc être simplement exprimée par

$$C(x) = a w_u^T(x) w_v(x) \tag{2.18}$$

avec a l'aire du triangle exprimée selon les coordonnées uv.

Au final, la matrice obtenue par Baraff et Witkin décrivant la topologie du maillage, possède les mêmes caractéristiques que celle présentée précédemment à partir des définitions de Volino et Magnenat-Thalmann. Cette matrice est diagonale et creuse. En effet, malgré des définitions d'interactions différentes, la notion de voisinage reste la même et donc la disposition des éléments non nuls de la matrice est identique.

2.3 Modèles continus

La simulation de textiles que nous avons implantée est basée sur un système à particules de type masses-ressorts. Mais nous pouvons noter qu'il existe d'autres types de modélisation basées non pas sur un modèle discret, mais sur un modèle continu [117, 91, 84].

Comme il n'est pas concevable de représenter chaque fil d'un textile par une arête du maillage, nous avons à choisir une certaine résolution de l'objet. Si nous ne voulons pas dépendre de cette résolution, une pièce du tissu doit pouvoir être représentée par un objet continu, autorisant l'utilisation de modèles à faible résolution sans pour autant perdre les propriétés de base du tissu.

Un modèle continu peut être dérivé en une discrétisation cohérente dans le sens où la solution calculée converge vers la solution exacte du modèle continu quand la résolution est augmentée. Ceci permet notamment le passage d'une résolution à une autre sans avoir à changer les propriétés du textile.

2.3.1 Équation du mouvement

Les mécanismes continus permettent de décrire et de modéliser des objets déformables. Les quantités de base utilisées pour des mécanismes continus sont *la déformation* ε, avec une déformation minimale notée ϵ et *la contrainte* σ, force proportionnelle à la longueur dans le cas des surfaces ou à l'aire dans le cas des volumes. Dans le cas d'un ressort à une dimension, ces entités sont des scalaires. La déformation du ressort est alors quantifiée par le rapport entre son élongation et sa longueur, et la contrainte représente la force du ressort. Dans le cas de surfaces ou de volumes, ces entités sont des tenseurs.

Dans le cas d'une élasticité linéaire, les relations entre les tenseurs de contrainte σ et de déformation ε sont supposées linéaires, et leurs dépendances sont formulées à partir d'un tenseur d'élasticité C. Ceci est formulé à partir de la loi de Hooke qui régit le comportement des solides élastiques :

$$\sigma_{ij} = C_{ijkl}\,\varepsilon_{kl}. \tag{2.19}$$

C est un tenseur de rang 4 symétrique contenant les propriétés du tissu. La loi de Hooke est utilisée pour calculer le tenseur de contrainte dans l'équation du mouvement d'un objet élastique continu :

$$\rho\frac{\partial^2 x}{\partial t^2} - div\ \sigma = f, \tag{2.20}$$

où ρ est la densité de masse et f la densité des forces externes (par exemple la densité de la force de gravité est ρg).

2.3.2 Calcul des déformations

Les surfaces sont plus compliquées à traiter qu'un simple ressort unidirectionnel, et les descriptions des déformations sont beaucoup plus complexes. Les textiles peuvent être vus comme des surfaces régulières. Les déformations des surfaces régulières dans $\mathbb{R}^3$ sont décrites par des tenseurs de déformation en respectant un état de référence indéformable. Dans cet état d'équilibre, noté **r**, l'objet n'est pas déformé et l'énergie d'élasticité est nulle. Cet état **r** est paramétré à l'intérieur d'un domaine $U \times V$. Sous la contrainte des forces, cet état se déforme pour arriver à un nouvel état **s**(u,v). Le déplacement entre ces deux états est défini par $\mathbf{d}(u,v) = \mathbf{s}(u,v) - \mathbf{r}(u,v)$ comme l'explique la figure 2.6.

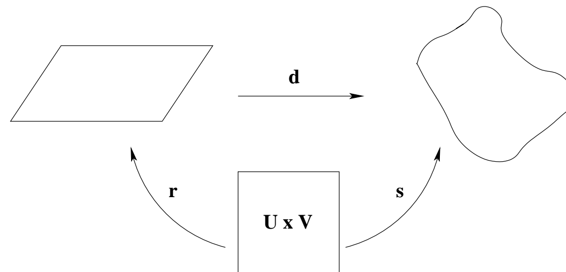


Figure 2.6 *La configuration de référence : l'état d'équilibre est défini par le déplacement **r** dans l'espace $U \times V$. Il est ensuite transformé par la déformation **d** en une nouvelle configuration (déformée), définie par le déplacement **s** dans l'espace $U \times V$.*

A partir de ces considérations, un tenseur de déformation non-linéaire ε est défini comme la différence entre l'état courant et l'état d'équilibre de l'objet.

2.3.3 Résolution de l'équation du mouvement

L'équation (2.20) est une Équation Différentielle Partielle (EDP) qui doit être résolue par rapport aux paramètres du temps et du domaine. Une procédure standard consiste à semi discrétiser le système dans l'espace avec des différences finies ou des éléments finis. Ceci permet de formuler cette équation sous la forme d'une simple Équation Différentielle Ordinaire dépendant du temps t et pouvant être résolue par n'importe quelle méthode d'intégration. Cette méthode permet donc de réduire une EDP du type (2.20) en une EDO du type (2.1).

Afin de simuler le comportement d'un tissu, les forces de flexion sont exprimées en tenant compte des propriétés volumiques du tissu. De plus nous pouvons dériver à partir de ce modèle continu un système à particules modélisant les objets continus avec une discrétisation par éléments finis [62, 64].

Nous avons actuellement implanté une simulation de textiles basée sur un système à particules. Nous avons en effet préféré partir d'un modèle discret, plus simple à implanter, pour concentrer nos efforts sur la recherche d'une parallélisation efficace des algorithmes de simulation. Mais il sera possible par la suite de modifier le modèle physique pour utiliser un modèle continu plus complexe mais n'impliquant pas de modification sur les choix de parallélisation. En effet, d'un point de vue algorithmique, il n'y a pas de différences notables entre l'implantation d'une simulation basée sur un modèle discret qu'une simulation basée sur un modèle continu qui se ramène au final à une discrétisation de l'objet. Les calculs sont simplement plus complexes à mettre en oeuvre.

2.4 Détection et traitement des collisions

Une simulation physique d'objets rigides ou non-rigides peut se diviser en trois étapes majeures. Ces étapes sont à résoudre aussi efficacement que possible mais également aussi précisément que nécessaire. Ces étapes sont : (1) le calcul de la dynamique des objets basée sur les forces et les interactions du maillage ; (2) la détection des collisions entre les objets de la scène et (3) le calcul du traitement des collisions, c'est-à-dire le calcul des forces engendrées par les collisions. Dans le cas particulier de la simulation de textiles, une étape importante consiste donc à déterminer les interactions ou les pénétrations du tissu avec lui-même ou son entourage.

Nous avons montré que l'utilisation d'un système à particules permet une modélisation réaliste du mouvement d'un tissu. De plus, cette modélisation a le mérite de reproduire des interactions réalistes entre les objets contenus dans le système. Le calcul de la trajectoire des particules est ensuite effectué en intégrant un système d'équations différentielles (phase 1). Les valeurs initiales de ce système, c'est-à-dire les positions et les vitesses initiales, étant fournies au départ de la simulation. Nous devons ensuite détecter les éventuelles collisions du tissu avec lui-même ou avec d'autres objets (phase 2) et les traiter en ajustant les vitesses et/ou les positions des particules, c'est-à-dire en modifiant les valeurs initiales du système (phase 3).

Durant ces dernières années, de nombreuses recherches ont été effectuées sur ce problème de la détection et le traitement des collisions [82, 98, 27, 28, 78, 92, 14]. Notons que Lin et Gottschalk [79] ont publié en 1998 un état de l'art sur cette question. L'utilisation de boîtes ou de sphères englobantes est très répandue pour diminuer la complexité relative à la vérification de l'interaction éventuelle d'un triangle avec chacun des autres triangles contenu dans le maillage [87]. Dans le cas où les objets sont des *polyèdres convexes*, des algorithmes rapides ont été développés travaillant en temps linéaire [56, 86]. D'autres méthodes surpassant la restriction de la convexité ont été développées, se basant sur des approches de volumes englobants hiérarchiques : des volumes sphériques [77], des po-

lytopes orientés (k-DOPs) [74] ou encore des boîtes englobantes orientées sont utilisées dans OBBTrees [53].

L'identification d'objets en collision peut être réalisée avec d'autres stratégies [29]. Certains préfèrent partitionner la scène en grilles régulières au lieu d'utiliser des boîtes englobantes [23, 122]. Ou encore d'autres travaux ont été effectués sur l'emploi du matériel graphique dans la détection des collisions [137].

Ces méthodes permettent d'obtenir des temps de réponse interactifs dans le cas de scènes avec plusieurs milliers de triangles. Mais dans le cas des simulations gouvernées par un système différentiel, il est nécessaire d'utiliser d'autres techniques plus sophistiquées pour obtenir du temps réel.

A l'heure actuelle, notre simulation de textiles ne comporte pas de phase de détection et de traitement de collisions. Mais l'implantation a été élaborée en ayant toujours à l'esprit la possibilité de rajouter facilement cette phase. Au niveau dynamique, la phase de traitement des collisions est équivalente à un ajout de forces exercées sur les particules. L'ajout de cette phase va donc augmenter le nombre de calculs à effectuer à chaque pas de temps de la simulation.

D'autre part nous avons vu qu'une des caractéristiques de ce type de simulation résidait dans la notion de voisinage : les particules sont reliées à d'autres particules via des ressorts, engendrant ainsi des dépendances entre les particules. Le traitement des collisions du tissu avec lui-même ou avec d'autres objets de la scène va modifier cette notion de voisinage tout en gardant le même ordre de grandeur : les dépendances entre les particules restent assez limitées.

Nous avons vu dans la section 2.2.3 que cette notion de voisinage détermine les structures de données de l'application. Lors de la phase de détection des collisions, les matrices des dérivées des forces vont être modifiées afin de prendre en compte les nouvelles interactions qui existent entre les particules ou encore les chocs qui ont pu se produire avec d'autres objets contenus dans la scène. Ces modifications se traduisent par l'ajout dans la matrice de nouveaux éléments en dehors de la diagonale. Mais ces nouveaux éléments ne sont pas en nombre suffisant pour remettre en cause la non densité de la matrice. Les caractéristiques de l'application ne seront donc pas modifiées par l'ajout de cette phase.

2.5 Conclusion

Pour effectuer une simulation de textiles avec une modélisation physique nous avons le choix entre deux types de modèles : un modèle discret ou un modèle continu. Les méthodes par éléments finis sont souvent privilégiées dans les cas où une grande précision est nécessaire. Par contre dans les cas où le but est simplement d'obtenir des résultats corrects avec un temps de calcul minimal, les méthodes à particules sont à privilégier. De plus, ces méthodes ont également le mérite d'être plus simples à implanter.

Notre simulation de textiles est basée sur un système de particules de type masses-ressorts. Les particules sont reliées entre elles par des ressorts afin de simuler au mieux les interactions qui existent entre elles et ainsi reproduire le comportement du tissu.

Après avoir calculé les forces appliquées sur chacune des particules, l'intégration de l'équation du mouvement de Newton permet de déterminer les états des particules (positions, vitesses). La formulation de l'équation du mouvement s'exprime sous la forme d'un système différentiel d'ordre 2, facilement transformable en un système différentiel d'ordre 1 plus facile à résoudre grâce à la multitude de méthodes existantes. Le chapitre suivant présente différentes méthodes d'intégration possibles pour résoudre ce système.

Après cette phase de calcul de la dynamique des particules, vient la détection des éventuelles collisions du textile avec lui-même ou avec d'autres objets contenus dans la scène. Cette phase engendre une mise à jour des positions des particules et une modification de leur voisinage. En résumé, la boucle de la simulation dynamique comporte les étapes de calcul suivantes :

Algorithme 1 Boucle de la simulation de textiles

```
1 :  Répéter
2 :     Forces exercees par les ressorts
3 :     Forces externes (gravite, vent)
4 :     Calculer les positions et vitesses par integration des accelerations
5 :     Tant Que (collisions ou auto-collisions) Faire
6 :         Rectifier positions et vitesses
7 :     Fin
8 :     Afficher
9 :  Fin
```

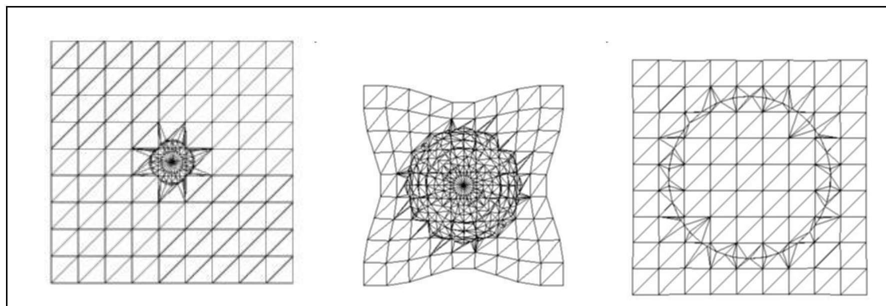


Figure 2.7 *Exemple d'utilisation d'un maillage adaptatif lors du traitement des collisions (images du projet Modeling of Virtual Textiles (MoViTex), extraites de leur site Web : http ://www.gris.uni-tuebingen.de/projects/physsim/).*

Notre objectif durant ce travail de recherche a été l'obtention d'une parallélisation efficace de cette boucle de simulation. C'est pourquoi nous avons choisi d'implanter dans

un premier temps une simulation de textiles basée sur un maillage fixe avec un modèle physique simplifié mais gardant toutes les propriétés intrinsèques à ce type de simulation, c'est-à-dire en préservant les dépendances de calculs dues aux interactions qui existent entre les particules.

En effet, la modélisation implantée est basée sur un maillage fixe avec un modèle discret de type masses-ressorts mais ne comportant que des ressorts longitudinaux et diagonaux. Il sera possible par la suite de complexifier cette simulation en utilisant une structure de maillage adaptative optimisant par exemple le traitement des collisions grâce à un raffinement de la zone concernée (figure 2.7) ou encore de rajouter les ressorts de flexion présentés dans le modèle de Provot sans aucune modification du point de vue parallélisme. Il sera même possible par la suite d'opter pour un modèle continu qui présente l'avantage de reproduire au mieux le comportement théorique du textile.

Méthodes d'intégration numérique 3

Nous avons choisi de modéliser notre tissu à l'aide d'un système de type masses-ressorts basé sur une discrétisation de l'objet en particules. Nous avons vu que la formulation de l'équation du mouvement associé à cet objet conduit à la résolution d'un système d'Équations Différentielles Ordinaires (EDO). Au sein de ce chapitre, nous allons nous intéresser aux différentes méthodes d'intégration qui existent permettant de résoudre ce système et ainsi d'obtenir les positions et les vitesses de chacune des particules.

3.1 Introduction

Un système à particules permet de modéliser un objet sous la forme d'un ensemble de masses. La surface du textile est représentée à partir de la géométrie définie entre des particules voisines. Le comportement mécanique du tissu est simulé grâce aux forces d'interaction qui existent entre les particules. Ces interactions dépendent essentiellement de la position et de la vitesse de ces particules. L'évolution du système est calculée numériquement en utilisant la loi fondamentale de la dynamique formant un système numérique d'Équations Différentielles Ordinaires (EDO), creux et de grande taille. Ce système doit être intégré numériquement, afin d'obtenir l'évolution du système mécanique au cours du temps, c'est-à-dire les différentes positions des particules évoluant à chaque pas de temps.

Il existe essentiellement deux catégories de méthodes d'intégration [58, 59, 60, 95] : les méthodes explicites (Euler explicite, Stöermer-Verlet/leapfrog, Runge-Kutta, midpoint explicite) et les méthodes implicites (Euler implicite, Rosenbrock, midpoint implicite). Les méthodes explicites calculent l'état du pas de temps suivant à partir d'une extrapolation directe des états précédents utilisant notamment des évaluations des dérivées en ces points. Les méthodes implicites déduisent quant à elles l'état du système du pas de

temps suivant à partir d'un système d'équations. Les méthodes explicites sont des méthodes locales tandis que les méthodes implicites sont des méthodes globales, prenant en considération le système entier. Il existe également des méthodes d'intégration d'ordre supérieur se servant de plusieurs évaluations afin de calculer des solutions d'ordre supérieur plus précises dans le cas où le pas de temps serait réduit. Et il existe de la même façon des méthodes d'intégration d'ordre inférieur qui réduisent le nombre d'évaluations afin de calculer de simples extrapolations d'ordre inférieur, rapides à calculer mais imprécises.

Il existe également des méthodes dédiées à la simulation de textiles. Nous pouvons par exemple citer le travail effectué par Desbrun *et al.* [37]. Ils ont en effet proposé un algorithme stable dédié à l'animation de textiles, utilisant une méthode hybride explicite/implicite. Nous pouvons noter qu'en 2001, Hauth et Etzmuss [62] ont effectué une analyse théorique sur l'exploitation des propriétés particulières relatives à la mécanique des objets déformables et sur la stabilité de plusieurs méthodes d'intégration numérique. Cette analyse a été approfondie lors d'un tutoriel présenté en 2002 [63], dans lequel ils établissent des comparaisons entre les différentes méthodes d'intégration. De même Volino et Magnenat-Thalmann [128] ont effectué des comparaisons sur les méthodes d'intégration numérique utilisées dans la simulation de textiles. Witkin et Baraff ont, quant à eux, regroupé en 1993 dans un tutoriel [130] les méthodes d'intégration numériques de base ainsi que leur implémentation.

Suite à cette multitude de méthodes d'intégrations numériques, plusieurs critères peuvent être considérés afin d'effectuer un choix pour l'utilisation de telle ou telle méthode [128] :

- la taille du problème c'est-à-dire le nombre de particules utilisées afin de décrire le système mécanique,
- la précision souhaitée, reflétant la tolérance numérique autorisée entre la solution calculée et l'évolution théorique du système estimée par rapport au modèle mécanique,
- le contexte de la simulation, c'est-à-dire si le souhait est d'obtenir une simulation très précise en prenant compte un maximum de facteurs mécaniques ou plutôt d'obtenir une simulation plausible,
- la rigidité du problème engendrée par les interactions entre les particules et la taille des intervalles de temps choisis, pouvant entraîner des problèmes d'instabilité ou de mauvaises précisions numériques,
- le temps mis pour calculer une itération de la simulation et le nombre de dérivations mécaniques (calcul des forces des particules à partir de leurs positions et de leurs vitesses) nécessaires à la méthode pour chaque itération,
- la convergence de la méthode sachant qu'une méthode d'intégration numérique converge si, quand le pas de temps tend vers zéro, les solutions numériques se confondent avec les solutions analytiques,
- la stabilité du schéma numérique analysée à partir du test de Dahlquist [32].

Ce chapitre se décompose en plusieurs parties. Il commence par la présentation de la

formulation abstraite de l'équation du mouvement appliquée au système mécanique. Elle s'exprime sous la forme d'un système d'Équations Différentielles Ordinaires d'ordre 1. Les différentes méthodes d'intégration que nous avons implantées seront ensuite analysées. Trois méthodes ont été intégrées : la méthode d'Euler explicite, la méthode d'Euler implicite et la méthode de Stöermer-Verlet ou leapfrog. Pour chacune de ces méthodes, nous verrons la formulation générale de son schéma d'intégration numérique, ainsi que son application dans le cadre particulier de l'animation de textiles. Nous verrons que l'application du schéma d'intégration d'Euler implicite conduit à la résolution d'un système linéaire creux. Cette résolution sera faite en employant la méthode du Gradient Conjugué qui est expliquée en détails dans le chapitre A de l'annexe. Enfin ce chapitre se terminera par une discussion des différentes méthodes d'intégration utilisées.

3.2 Forme abstraite du système différentiel

Dans le chapitre précédent, nous avons vu que la formulation de l'équation du mouvement pouvait s'exprimer sous la forme d'un système d'équations différentielles, initialement du second ordre mais facilement transformable en un système du premier ordre, dans lequel les dérivées sont établies par rapport au temps t :

$$\begin{pmatrix} v(t) \\ x(t) \end{pmatrix}' = \begin{pmatrix} M^{-1} f\left(t, x(t), v(t)\right) \\ v(t) \end{pmatrix}, \begin{pmatrix} v(t_0) \\ x(t_0) \end{pmatrix} = \begin{pmatrix} v_0 \\ x_0 \end{pmatrix}.$$

Nous allons présenter dans ce chapitre quelques méthodes d'intégration possibles pour résoudre ce système différentiel permettant ainsi l'obtention des vitesses et positions des particules constituant le tissu. Pour cela il convient de définir ce système d'EDO sous une forme plus abstraite :

$$\boxed{y'(t) = f\left(t, y(t)\right), y(t_0) = y_0.} \tag{3.1}$$

3.3 Méthode d'Euler explicite

3.3.1 Schéma d'intégration numérique

Le schéma d'Euler explicite sert à calculer l'évolution de l'état d'un système dans le temps. C'est la méthode numérique la plus simple et la plus ancienne. Le temps est décomposé en intervalles de longueur h. Connaissant la solution au temps t, nous cherchons à l'évaluer au temps $t + h$, c'est-à-dire à l'évaluer après un avancement dans le temps correspondant à l'intervalle de temps h. Pour cela nous remplaçons dans le système (3.1) la dérivée y'(t) par son approximation mathématique :

$$\frac{y(t+h) - y(t)}{h} \approx y'(t) = f(t, y(t)). \tag{3.2}$$

A partir ce cette formulation, nous en déduisons la formule du schéma d'intégration numérique explicite d'Euler pour un simple avancement h dans le temps :

$$y(t+h) = y(t) + hf(t, y(t)). \tag{3.3}$$

3.3.2 Interprétation géométrique

En itérant cette méthode, nous obtenons une séquence d'approximations numériques $Y_n \approx y(t_n) = y(t_0 + nh)$. La figure 3.1 montre que d'un point de vue géométrique, cette méthode revient à calculer la tangente à la solution au temps t afin d'obtenir une approximation de la solution au temps $t + h$.

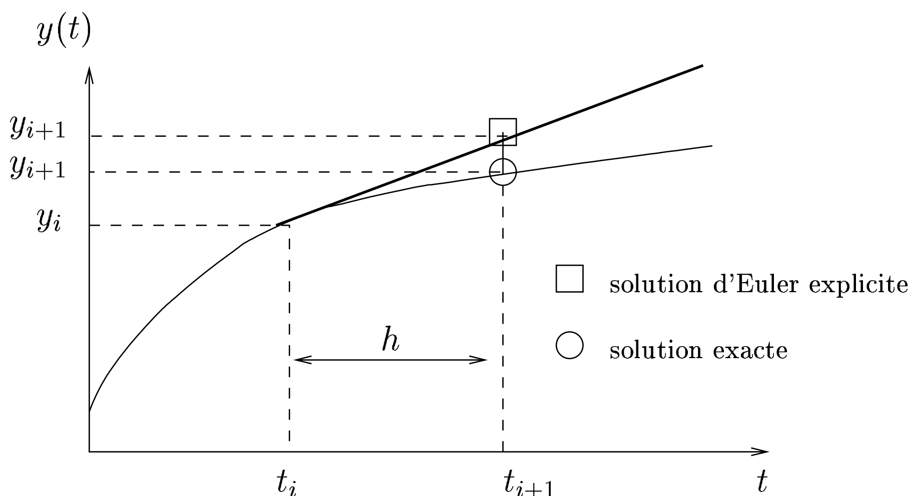


Figure 3.1 *Interprétation géométrique du schéma d'intégration d'Euler explicite : l'approximation de la solution au temps $t + h$ est obtenue en calculant la tangente à la solution au temps t.*

3.3.3 Convergence

Une méthode d'intégration numérique converge si, quand $h \to 0$, les solutions numériques Y_n se confondent avec les solutions analytiques. Toutes les méthodes utilisées doivent converger, c'est pourquoi nous ne discutons pas ici de méthodes non-convergentes ou encore du critère de convergence, mais nous nous intéressons à la précision ou à l'ordre d'intégration d'une méthode. C'est-à-dire que nous étudions les vitesses de convergence des méthodes quand $h \to 0$ ou encore la précision des solutions obtenues pour un h donné. Nous rappelons que le développement de Taylor associé à $y(t+h)$ s'exprime sous

la forme :

$$y(t+h) = y(t) + hy'(t) + \frac{h^2}{2!}y''(t) + ... + \frac{h^n}{n!}\frac{\partial^n y}{\partial t^n} + ... \tag{3.4}$$

En utilisant le développement de Taylor pour la solution exacte après un seul intervalle de temps

$$y(t+h) = y(t) + hy'(t) + \frac{h^2}{2}y''(t) + \mathcal{O}\left(h^3\right), \tag{3.5}$$

nous trouvons pour l'approximation numérique Y_1 produite par un pas d'Euler explicite

$$y(t_1) - Y(t_1) = \mathcal{O}\left(h^2\right). \tag{3.6}$$

Si nous poursuivons la méthode en utilisant la solution numérique Y_1 comme valeur de départ pour l'intervalle de temps suivant, nous obtenons une erreur globale d'ordre h

$$y(t_n) - Y(t_n) = \mathcal{O}\left(h\right). \tag{3.7}$$

Cela signifie que la méthode d'Euler explicite converge linéairement ou encore que cette méthode est *d'ordre 1*.

3.3.4 Stabilité

Nous reprenons dans cette partie, l'analyse de stabilité effectuée par Michael Hauth et Olaf Etzmuss [62, 63].

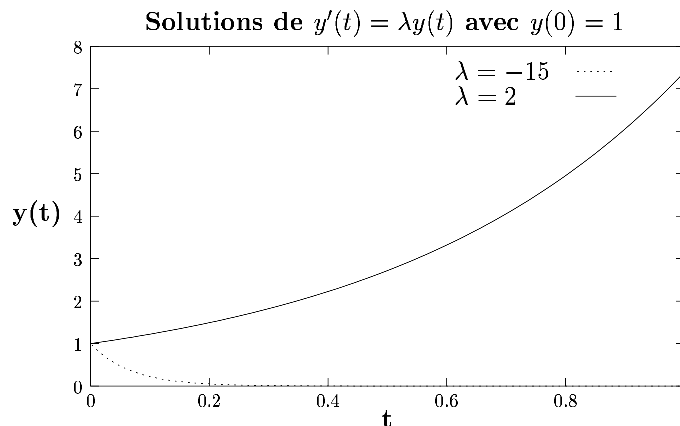


Figure 3.2 *Solutions du système différentiel défini par* $y'(t) = \lambda y(t), y(0) = 1$ *avec* $\lambda = -15$ *et* $\lambda = 2$*, pour* $t \in [0, 1]$

Considérons l'*équation test de Dahlquist*, qui est utilisée comme outil d'évaluation et de compréhension de la stabilité des méthodes d'intégration [32, 95]. Elle est définie par :

$$y'(t) = \lambda y(t), \lambda \in \mathbb{C}. \tag{3.8}$$

La solution exacte de cette équation de valeur initiale $y(0) = y_0$, est donnée par

$$y(t) = e^{\lambda t} y_0.$$

La figure 3.2 représente les solutions de cette équation pour $\lambda = -15$ et $\lambda = 2$ avec $t \in [0, 1]$.

Dans le cas amorti caractérisé par la partie réelle du complexe λ négative, c'est-à-dire quand $\mathcal{R}e\lambda < 0$, la convergence ne peut être obtenue que par l'utilisation d'intervalles de temps très petits [63]. Dans ce cas, l'exposant étant négatif, la solution analytique est bornée pour $t \to \infty$. Une méthode numérique est alors nécessaire afin d'obtenir une solution bornée. D'autre part un schéma d'intégration procurant une solution bornée est dit *stable*.

Si nous appliquons le schéma d'Euler explicite à l'équation (3.8) avec un pas fixé de taille h, le $n^{ème}$ point de chaque solution est donné par :

$$Y_n = (1 + h\lambda)^n y_0. \tag{3.9}$$

Cette solution numérique est bornée et donc stable si et seulement si $|1 + h\lambda| < 1$, c'est-à-dire pour $h\lambda$ compris dans le cercle unitaire de centre $(-1, 0)$ dans le plan complexe comme l'illustre la figure 3.3.

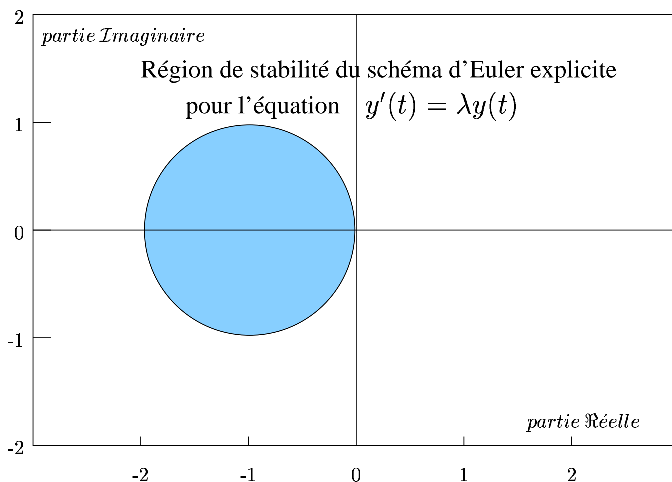


Figure 3.3 *Région de stabilité de la méthode d'Euler explicite pour l'équation $y'(t) = \lambda y(t)$*

Les solutions numériques ne peuvent donc converger que si la taille du pas de temps est en dessous d'une certaine limite dictée par λ, à savoir par $h < \lambda^{-1}$ dans le cas présent. Si l'amortissement est augmenté, c'est-à-dire si $\mathcal{R}e\lambda \to -\infty$, nous devons avoir $hv \to 0$ pour assurer la stabilité de la solution obtenue par la méthode d'Euler explicite. Ceci signifie que le pas de temps est artificiellement limité et qu'il ne peut pas dépasser une limite de stabilité.

3.3.5 Application à la simulation de textiles

Maintenant que le schéma d'intégration numérique de la méthode d'Euler explicite a été établi, il peut être appliqué au système différentiel obtenu pour la simulation de textiles. Il va ainsi permettre l'obtention des positions et des vitesses de chacune des particules du système. Les accélérations, quant à elles, sont données en appliquant la loi fondamentale de la dynamique. Les formules suivantes seront donc utilisées pour les temps t_0 et $t + h$:

$$\left\{ \begin{array}{lcl} v'(t_0) & = & M^{-1} f\left(t_0, x(t_0), v(t_0)\right) \\ v(t_0) & = & v_0 \\ x(t_0) & = & x_0 \end{array} \right. , \quad \left\{ \begin{array}{lcl} v'(t) & = & M^{-1} f\left(t, x(t), v(t)\right) \\ v(t+h) & = & v(t) + hv'(t) \\ x(t+h) & = & x(t) + hv(t) \end{array} \right. .$$

Ce schéma d'intégration numérique est très facile à implanter, mais nous devons limiter la taille de l'intervalle de temps h pour garantir sa stabilité.

3.4 Méthode de Stöermer-Verlet/leapfrog

3.4.1 Schéma d'intégration numérique

Nous allons nous intéresser à une seconde méthode appelée leapfrog ou Stöermer-Verlet. Considérons le système d'Équations Différentielles Ordinaires (2.6) donné au second ordre

$$x''(t) = M^{-1} f(t, x(t)), \tag{3.10}$$

c'est-à-dire avec $M^{-1} f(t, x(t), x'(t)) = M^{-1} f(t, x(t)$. En effet le schéma n'est pas applicable aux systèmes du premier ordre de la forme (3.1).

Pour le dériver, nous allons approximer la vitesse v au temps $t + (2i + 1)h/2$ et la position x au temps $t + ih$ en centrant les différences, c'est-à-dire en utilisant des grilles échelonnées représentées par la figure 3.4.

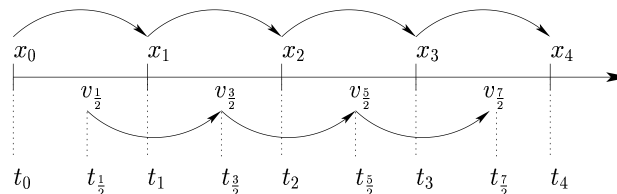


Figure 3.4 *Grilles échelonnées pour la méthode d'intégration de Störmer-Verlet/leapfrog*

Nous obtenons

$$\frac{v_{n+1/2} - v_{n-1/2}}{h} = f(x_n), \tag{3.11}$$

$$\frac{x_{n+1} - x_n}{h} = v_{n+1/2}, \tag{3.12}$$

ce qui amène à

$$v_{n+1/2} = v_{n-1/2} + hf(x_n), \tag{3.13}$$

$$x_{n+1} = x_n + hv_{n+1/2}. \tag{3.14}$$

Cette méthode est d'ordre 2 comme nous pouvons le voir en substituant (3.13) dans (3.14)

$$\frac{x_{n+1} - 2x_n + x_{n-1}}{h} = f(x_n). \tag{3.15}$$

De cette équation, nous pouvons en déduire une autre façon de formuler le schéma de Verlet

$$v_n - v_{n-1} = hf(x_n), \tag{3.16}$$

$$x_{n+1} - x_n = hv_n, \tag{3.17}$$

qui omet l'utilisation des grilles échelonnées et des demi pas de temps employées précédemment.

Pour les cas où le système différentiel ne peut être exprimé sous la forme (3.10), nous pouvons remplacer $f(x_n)$ par $f(x_n, v_{n-1})$ aux dépends d'un peu de stabilité. Normalement nous devrions le remplacer par $f(x_n, v_n)$ mais nous retomberions alors dans le cas d'une méthode implicite.

3.4.2 Stabilité

La méthode Stöermer-Verlet/leapfrog n'est pas inconditionnellement stable [63], c'est-à-dire qu'il y a des restrictions à appliquer sur l'intervalle de temps afin d'assurer sa stabilité. En effet cette méthode fournit seulement une solution bornée pour un intervalle de temps $h > 0$ pour des équations différentielles ordinaires d'ordre 2 de la forme (3.10), c'est-à-dire sans terme d'amortissement. C'est pourquoi cette méthode est dite *conditionnellement stable*. Kačić-Alesić *et al.* [71] montrent que cette stabilité est régie par la taille du pas de temps, c'est-à-dire que la méthode Stöermer-Verlet/leapfrog est stable sous les conditions suivantes :

$$\Delta t \leq \frac{2}{\omega_d}, \tag{3.18}$$

où ω_d représente la plus haute pulsation du système. Pour un unique ressort non amorti de raideur k et de masse m, sa pulsation propre est définie par :

$$\omega_{d0} = \sqrt{\frac{k}{m}}, \tag{3.19}$$

par contre pour un ressort amorti, elle est définie par :

$$\omega_d = \sqrt{\omega_{d0}^2 - \left(\frac{\nu}{2m}\right)^2}, \quad m = \frac{1}{\frac{1}{m_a} + \frac{1}{m_b}}$$

avec m sa masse harmonique, et m_a et m_b les masses des deux extrémités du ressort. L'amortissement critique ν_c correspond au facteur d'amortissement du système quand il n'y a plus d'oscillation, c'est-à-dire pour $\omega_d = 0$, soit $\nu_c = 2\sqrt{k\,m}$.

Le pire scénario possible correspond au cas où les n ressorts constituant le système sont connectés en parallèle entre deux particules. La raideur du ressort équivalent à la combinaison de tous les ressorts est donnée par $K = \sum_{i=1}^{n} k_i$. Nous pouvons en déduire que l'intégration est stable si

$$\Delta t \leq 2\sqrt{\frac{m_{min}}{K}}, \tag{3.20}$$

avec m_{min} la plus petite masse de l'ensemble des ressorts contenus dans le système. Mais Kaĉić-Alesić *et al.* [71] remarquent qu'en pratique cette règle n'est pas souvent utilisée en faveur de la "règle des 10%" décrite par Provot (cf. section 2.2.1) qui est plus facile à employer.

Le problème de la stabilité d'une méthode d'intégration peut également être abordée d'une autre manière [71] : étant donné un pas de temps fixé, quelle force les ressorts peuvent exercer avant de déstabiliser le système ? En considérant le même pire scénario que précédemment et en supposant que tous les ressorts ont la même raideur k_{max}, nous avons alors $K = \sum_{i=1}^{n} k_i = n\ k_{max}$. L'intégration est alors stable si

$$k_{max} \leq \frac{4\ m_{min}}{n\ \Delta t^2}. \tag{3.21}$$

D'autre part, nous pouvons noter que la méthode Stöermer-Verlet/leapfrog est *symétrique* par rapport au temps, impliquant une propriété géométrique importante qui est la *réversibilité*. Le fait d'inverser la direction de la vitesse initiale ne modifie pas la trajectoire obtenue, mais inverse juste la direction du mouvement.

De plus cette méthode a la propriété de conserver l'énergie totale du système. Pour plus de détails sur les propriétés spécifiques à cette méthode, le lecteur peut se référer à l'article de Hairer, Lubich et Wanner [57] publié en 2003.

3.4.3 Application à la simulation de textiles

Nous avons étudié le schéma de la méthode de Stöermer-Verlet/leapfrog en approximant la position x au temps $t + ih$ et la vitesse v au temps $t + (2i+1)h/2$. Afin de rendre plus intuitif son utilisation lors du schéma itératif de la simulation, nous la présentons ici

au temps $t + h$ pour la position x et au temps $t + \frac{h}{2}$ pour la vitesse v. L'accélération est quant à elle toujours calculée à partir de la loi fondamentale de la dynamique :

$$\left\{ \begin{array}{lcl} v'(t_0) & = & M^{-1} f(t_0, x(t_0), v(t_0)) \\ v(t_0) & = & v_0 \\ v(\frac{h}{2}) & = & v_0 + \frac{h}{2} v'_0 \\ x(t_0) & = & x_0 \end{array} \right. , \quad \left\{ \begin{array}{lcl} v'(t) & = & M^{-1} f(t, x(t), v(t)) \\ v(t + \frac{h}{2}) & = & v(t - \frac{h}{2}) + h v'(t) \\ x(t + h) & = & x(t) + h v(t + \frac{h}{2}) \end{array} \right. .$$

Ce schéma d'intégration est aussi facile à implanter que celui d'Euler explicite, mais expérimentalement il s'est révélé réellement plus stable, permettant notamment l'utilisation de pas de temps plus grands.

3.5 Méthode d'Euler implicite

3.5.1 Schéma d'intégration numérique

Les méthodes explicites peuvent se révéler instables, c'est pourquoi nous avons étudié une autre méthode répondant mieux à nos attentes. Pour cela nous reconsidérons le système différentiel (3.1) dans lequel nous introduisons $y(t + h)$. Nous obtenons alors

$$\frac{y(t + h) - y(t)}{h} \approx y'(t + h) = f(t + h, y(t + h)), \tag{3.22}$$

permettant d'en déduire

$$y(t + h) = y(t) + h f(t + h, y(t + h)). \tag{3.23}$$

Nous obtenons alors la formule d'intégration

$$Y_{n+1} = Y_n + h f(t + h, Y_{n+1}), \tag{3.24}$$

appelée méthode d'Euler implicite. Comme sa variante explicite, cette méthode est également d'ordre 1. La solution numérique est seulement donnée implicitement comme la solution de l'équation non-linéaire

$$Y_{n+1} - h f(t + h, Y_{n+1}) - Y_n = 0. \tag{3.25}$$

3.5.2 Stabilité

Si nous appliquons ce schéma d'intégration pour résoudre l'équation test de Dahlquist [62, 63], nous obtenons la formule de récurrence

$$Y_n = (1 - h\lambda)^{-n} y_0. \tag{3.26}$$

Cette solution numérique est bornée dans le cas où $|(1 - h\lambda)^{-1}| < 1$, c'est-à-dire pour $h\lambda$

non compris dans le cercle unitaire de centre $(1, 0)$ comme l'illustre la figure 3.5.

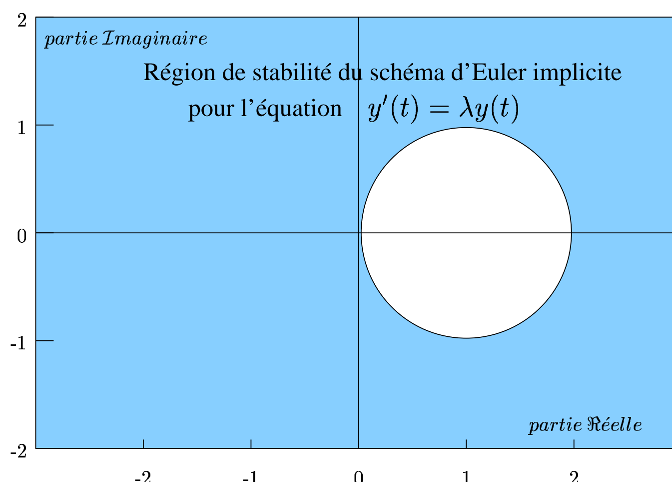


Figure 3.5 *Région de stabilité de la méthode d'Euler implicite pour l'équation* $y'(t) = \lambda y(t)$

Si nous supposons que $\lambda < 0$, la relation $|(1 - h\lambda)^{-1}| < 1$ est vérifiée quelque soit $h > 0$. Il n'y a donc aucune restriction sur l'intervalle de temps h, cette méthode est alors dite *inconditionnellement stable*.

3.5.3 Application à la simulation de textiles

Nous appliquons ici la méthode d'Euler implicite à notre système de particules. Il est important de noter que les travaux effectués par Baraff et Witkin [13] ainsi que ceux effectués par Volino et Thalmann [126] ont permis de prouver que les méthodes d'intégration implicites utilisées pour résoudre des équations différentielles autorisent l'utilisation d'intervalles de temps assez importants sans pour autant perdre en stabilité numérique. En effet à l'époque de ces publications, les méthodes explicites étaient largement utilisées en animation mais avec la contrainte d'utiliser des pas de temps très petits. Nous reprenons par ailleurs dans cette partie le raisonnement fait par David Baraff et Andrew Witkin [13] afin de simplifier le système obtenu lors de l'emploi du schéma d'intégration d'Euler implicite.

Supposons que la position $x(t)$ et la vitesse $v(t)$ du système soient connues au temps t, le but est de déterminer la nouvelle position $x(t + h)$. Nous reprenons le système différentiel déjà défini en (2.7) que nous rappelons ici :

$$\begin{pmatrix} v(t) \\ x(t) \end{pmatrix}' = \begin{pmatrix} M^{-1} f\left(t, x(t), v(t)\right) \\ v(t) \end{pmatrix}. \tag{3.27}$$

Afin de simplifier les notations, nous définissons les variables Δv et Δx telles que :

$$\left\{ \begin{array}{lcl} \Delta v & = & v(t+h) - v(t), \\ \Delta x & = & x(t+h) - x(t). \end{array} \right. \tag{3.28}$$

Si nous appliquons le schéma d'Euler implicite (3.23) au système (3.27) nous obtenons alors :

$$\begin{pmatrix} \Delta v \\ \Delta x \end{pmatrix} = h \begin{pmatrix} M^{-1} f\left(x(t) + \Delta x, v(t) + \Delta v\right) \\ v(t) + \Delta v \end{pmatrix}. \tag{3.29}$$

La méthode d'Euler implicite nécessite donc la résolution du système non-linéaire (3.29) afin d'obtenir les valeurs de Δx et Δv vérifiant ces équations. Nous utilisons alors les séries de Taylor appliquées à f, nous donnant l'approximation du premier degré

$$f\left(x(t) + \Delta x, v(t) + \Delta v\right) = f(t) + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial v}\Delta v. \tag{3.30}$$

dans laquelle les dérivées $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial v}$ devront être évaluées . En substituant cette approximation dans le système (3.29), nous obtenons le système linéaire suivant :

$$\begin{pmatrix} \Delta v \\ \Delta x \end{pmatrix} = h \begin{pmatrix} M^{-1}\left(f(t) + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial v}\Delta v\right) \\ v(t) + \Delta v \end{pmatrix}. \tag{3.31}$$

Nous ne considérons plus que la première équation du système (3.31) dans lequel nous remplaçons Δx par $h\left(v(t) + \Delta v\right)$ pour obtenir :

$$\Delta v = hM^{-1}\left(f(t) + \frac{\partial f}{\partial x} h\left(v(t) + \Delta v\right) + \frac{\partial f}{\partial v}\Delta v\right). \tag{3.32}$$

Soit I la matrice identité, nous en déduisons le système

$$\left(I - hM^{-1}\frac{\partial f}{\partial v} - h^2 M^{-1}\frac{\partial f}{\partial x}\right)\Delta v = hM^{-1}\left(f(t) + h\frac{\partial f}{\partial x}v(t)\right), \tag{3.33}$$

qui peut également s'écrire sous la forme

$$\left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)\Delta v = h\left(f(t) + h\frac{\partial f}{\partial x}v(t)\right). \tag{3.34}$$

Il ne reste plus qu'à résoudre ce système afin d'obtenir Δv et ainsi calculer facilement

$$\left\{ \begin{array}{lcl} v(t+h) & = & v(t) + \Delta v, \\ x(t+h) & = & x(t) + hv(t+h). \end{array} \right. \tag{3.35}$$

Étapes de la méthode d'intégration d'Euler implicite

En résumé, pour obtenir les états des particules du système masses-ressorts en utilisant la méthode d'Euler implicite pour l'intégration du système différentiel, il faut :

1. Évaluer $f(t)$,
2. évaluer $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial v}$,
3. construire le système d'équations linéaires creux (3.34),
4. résoudre ce système pour obtenir Δv,
5. et enfin mettre à jour x et v.

L'utilisation d'une méthode implicite entraîne donc une quantité de calcul et une complexité importante. Elles résultent de la résolution de systèmes linéaires de grosses tailles impliquant généralement l'implantation d'algorithmes traitant des matrices creuses. Dans notre cas la difficulté principale réside dans la résolution du système (3.34) de la forme $A\Delta v = b$, où la matrice A est généralement une matrice creuse définie positive.

Pour résoudre ce système linéaire, nous pouvons appliquer la méthode du gradient conjugué [13, 127, 109, 75], permettant de tirer profit du fait que la matrice soit creuse. Le choix de cette méthode est motivée par sa rapidité de convergence par rapport à d'autres méthodes itératives. D'autre part, d'un point de vue algorithmique, la méthode du Gradient Conjugué a l'avantage de n'utiliser que les opérations de base de l'algèbre linéaire à savoir le produit d'une matrice par un vecteur. Ces opérations peuvent être simples à implanter à condition de choisir les structures de données appropriées permettant de gérer la non densité de nos données.

Méthode du Gradient Conjugué

Nous avons vu que l'application du schéma d'intégration d'Euler implicite sur le système à particules conduisait à la résolution du système

$$\left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)\Delta v = h\left(f(t) + h\frac{\partial f}{\partial x}v(t)\right)$$

permettant de trouver les vitesses des particules. Nous pouvons observer que ce système linéaire est de la forme $Ax = b$ avec A une matrice symétrique définie-positive, si nous définissons la matrice A, le vecteur solution x et le vecteur b par

$$A = \left(M - h\tfrac{\partial f}{\partial v} - h^2\tfrac{\partial f}{\partial x}\right), \quad x = \Delta v \quad b = \left(hf(t) + h^2\tfrac{\partial f}{\partial x}v(t)\right). \tag{3.36}$$

Il existe une multitude de méthodes permettant la résolution d'un tel système. Ces méthodes sont généralement répertoriées en deux catégories à savoir les méthodes *directes* et les méthodes *itératives*. Les méthodes directes trouvent la solution exacte en un nombre

fini d'opérations généralement d'ordre N^3, si N est la taille des vecteurs b et x et A est une matrice de taille $N \times N$. Les méthodes itératives, quant à elles, ne permettent pas d'obtenir la solution exacte du système $Ax = b$ en un temps fini, mais elles convergent asymptotiquement vers une solution. Néanmoins les méthodes itératives génèrent souvent une solution ayant une précision correcte, après un nombre relativement petit d'itérations et sont alors préférables aux méthodes directes. Ceci est généralement le cas quand N est important. De plus les méthodes itératives nécessitent moins de mémoire que les méthodes directes dans le cas où A est une matrice creuse.

Nous avons choisi d'employer la méthode du Gradient Conjugué pour résoudre le système linéaire. L'algorithme du Gradient Conjugué itère jusqu'à ce que le facteur d'erreur prenne une valeur en dessous de ε qui traduit la précision souhaitée [109, 75] :

Algorithme 2 Algorithme du Gradient Conjugué pour la résolution de $Ax = b$

1 : $\beta \leftarrow 0;$ `// Initialisation du facteur d'erreur`
2 : $x \leftarrow 0;$ `// Initialisation de la solution`
3 : $r \leftarrow b - Ax;$ `// Initialisation du vecteur de residu`

4 : **Faire**
5 : $\alpha \leftarrow r^T r;$ `// Initialisation du pas`

6 : **Si** $(\beta \neq 0)$
7 : $d \leftarrow r + (\frac{\alpha}{\beta})d;$ `// Calcul de la nouvelle direction`
8 : **Sinon**
9 : $d \leftarrow r;$ `// Initialisation du vecteur de direction`

10 : $\beta \leftarrow d^T Ad;$ `// Calcul du facteur d'erreur`
11 : $r \leftarrow r - (\frac{\alpha}{\beta})Ad;$ `// Calcul du vecteur de residu`
12 : $x \leftarrow x + (\frac{\alpha}{\beta})d;$ `// Calcul de la solution`
13 : $\beta \leftarrow \alpha;$ `// Nouveau facteur d erreur`

14 : **Jusqu'a** $(\beta < \varepsilon)$ `// Iteration jusqu'a la precision souhaitee`

En annexes, le chapitre A présente en détails cette méthode itérative particulièrement bien adaptée pour la résolution de systèmes linéaires creux impliquant des structures de données creuses. Nous avons seulement repris ici l'algorithme séquentiel de la méthode sur lequel nous nous sommes basé avant d'implanter une version parallèle dans notre simulation.

3.6 Conclusion

Un système à particules de type masses-ressorts permet de décrire la mécanique du textile en le discrétisant en un ensemble de masses interagissant entre elles par l'intermédiaire de ressorts plus ou moins complexes. La formulation de l'équation du mouvement dictée par la loi fondamentale de la dynamique, entraîne la résolution d'un système d'Équations Différentielles Ordinaires initialement d'ordre 2 mais facilement transformable en un système différentiel d'ordre 1. De nombreuses méthodes d'intégration sont relatées dans la littérature permettant la résolution d'un tel système.

Il y a donc un certain nombre de considérations à prendre en compte dans le choix d'une méthode d'intégration. En effet dans le cas des méthodes explicites, la précision des calculs du mouvement de toutes les particules est coûteuse, demandant beaucoup de ressources en calcul. Mais le résultat peut être extrêmement précis grâce à l'utilisation de méthodes d'ordres élevés décrivant complètement l'évolution du système.

Dans le cadre des méthodes implicites, la méthode d'Euler étant d'ordre faible, les solutions qu'elle fournit ne sont pas forcément précises malgré sa stabilité, en particulier lors de l'utilisation de grands pas de temps. Elle assure seulement une stabilité en estimant l'état d'équilibre du système et en convergeant vers cette estimation quand le pas de temps devient grand. De nombreux effets dynamiques disparaissent dans ces approximations entraînant l'utilisation de constantes d'amortissement artificiellement élevées. Par exemple dans le cas d'un tissu rigide rectangulaire suspendu, la méthode implicite va amener efficacement les sommets à leurs états d'équilibre en respectant les interactions dues aux liaisons entre particules, mais ne reproduira pas l'effet de drapé résultant des forces exercées par la gravité si un maillage raffiné et des grands pas de temps sont utilisés. En effet quand des déformations globales complexes sont impliquées, l'évolution du système non-linéaire complexe vers son état d'équilibre ne peut être calculé simplement à partir des dérivées linaires de son état courant.

Dans le cadre de notre simulation, nous avons limité nos expérimentations à trois méthodes. Notre premier choix s'est porté sur la méthode la plus classique : la méthode d'Euler explicite. Cette méthode a l'avantage d'avoir un schéma d'intégration très simple. Par contre elle est très instable et nécessite des pas de temps très petits pour rester stable, entraînant un très grand nombre d'itérations à calculer pour un temps d'animation donné.

C'est pourquoi nous avons ensuite opté pour la méthode Stöermer-Verlet/leapfrog, aussi facile à intégrer mais expérimentalement plus stable que la méthode d'Euler. D'autre part Provot [97] mettait en évidence dans sa thèse que cette méthode d'Euler modifié est de loin la plus performante en terme de rapport stabilité/coût en comparaison des méthodes de Runge-Kutta (midpoint, Runge-Kutta d'ordre 4). Pour cela il présente une comparaison des temps de calcul des différentes méthodes d'intégration explicites. Les tests sont établis pour une pièce de tissu carré de $0.4\ m$ de côté, suspendu par un seul point, de masse surfacique de $500g/m^2$. Son maillage est constitué d'un réseau de 20

$\times$ 20 noeuds et les ressorts au sein de ce système possèdent une raideur de $2N/m$. La période propre du système est d'environ de $\frac{1}{32}$ s. Le pas de temps du système devant être obligatoirement inférieure à cette période pour pouvoir rendre compte correctement des oscillations. Pour chaque pas de temps, il règle le pas de temps Δt de manière à être à la limite de la stabilité de l'intégration pour une durée totale de $10 \times \frac{1}{25}$ s d'animation. Ses résultats sont reproduits dans le tableau 3.1.

	Méthodes d'intégration			
	Euler	*midpoint*	*Runge-Kutta 4*	*leapfrog*
Δt (s)	$\frac{1}{3400}$	$\frac{1}{417}$	$\frac{1}{250}$	$\frac{1}{185}$
CPU (s)	16.06	3.81	4.31	1.09

TAB. 3.1 *Comparaison des méthodes d'intégration explicites [97] pour une pièce de tissu de période propre* $T_0 \simeq \frac{1}{32}s$

Même si les temps CPU datent de 1997, ils permettent de mettre en évidence les différences de temps de calcul qu'il existe entre ces quatre méthodes explicites. La méthode d'Euler classique nécessite, comme nous l'avions déjà vu d'un point de vue théorique, des pas de temps très faibles afin de garder un système stable. Cela entraîne pour une période de temps fixée, un plus grand nombre d'itérations à calculer et ainsi un temps de calcul nettement supérieur aux autres méthodes. La méthode du midpoint, nécessitant deux évaluations des forces, est beaucoup plus stable avec un pas de temps dix fois supérieur à celui de la méthode d'Euler. La méthode de Runge-Kutta d'ordre 4 est encore plus stable avec un pas de temps 1.7 fois supérieur à celui du midpoint, mais elle nécessite 4 évaluations des forces. Enfin nous pouvons observer que la méthode de Stöermer-Verlet/leapfrog est encore plus stable. D'autre part cette méthode est moins coûteuse en terme de calcul ne nécessitant comme la méthode d'Euler, qu'une seule évaluation des forces. C'est pourquoi Provot estime dans sa thèse que cette méthode est celle qui possède le meilleure compromis stabilité/coût.

Concernant les méthodes implicites, nous avons également opté pour la plus classique d'entre elles, à savoir la méthode d'Euler implicite. Les méthodes implicites nécessitent un grand nombre de calculs supplémentaires par rapport aux méthodes explicites, avec notamment la résolution d'un système linéaire creux de grande taille. Mais l'emploi de la méthode d'Euler implicite permet d'utiliser des pas de temps plus grands. La résolution du système linéaire s'effectue à l'aide de la méthode du Gradient Conjugué, permettant de tirer profit de la faible densité des matrices (les seuls éléments non nuls représentant les liaisons entre les particules).

Sources de parallélisme 4

L'objectif de cette thèse est l'obtention d'une plate-forme permettant de coupler une simulation parallèle d'objets physiques avec une visualisation sur plusieurs écrans. Pour atteindre ce but, il est nécessaire de mettre en évidence les caractéristiques de ce type de simulation. Pour cela nous avons étudié le cas particulier de la simulation de textiles. Cette analyse est ensuite extensible à d'autres simulations ayant les mêmes caractéristiques. Au sein de ce chapitre nous allons récapituler ces propriétés pour ensuite faire ressortir toutes les sources de parallélisme de ce type de simulations physiques.

4.1 Introduction

L'objectif de cette thèse est l'obtention d'une plate-forme permettant de coupler une simulation numérique parallèle d'objets déformables avec une visualisation sur plusieurs écrans. Nous nous intéressons essentiellement à des simulations basées sur une modélisation physique des objets de la scène. Pour atteindre ce but, il est nécessaire de mettre en évidence les caractéristiques de ce type de simulation. Pour cela nous avons étudié le cas particulier de la simulation de textiles.

Dans le chapitre 2, nous avons vu que deux types de modélisation physique sont usuellement employés en simulation de textiles : une modélisation continue ou une modélisation discrète. L'utilisation d'un modèle continu entraîne une plus grande complexité des calculs, mais elle se ramène au final à une discrétisation de l'objet. Les objets à simuler sont représentés sous la forme d'un *maillage* de particules. Sa topologie permet de connaître le *voisinage* de chacune des particules. Cette connaissance est indispensable pour pouvoir calculer les forces appliquées sur ces particules. Ces forces permettront par la suite de calculer les accélérations des particules. Puis l'intégration de l'équation

du mouvement de Newton permettra l'obtention des vitesses et positions des particules. Dans le chapitre 3 nous avons vu en détails les trois méthodes d'intégration implantées : la méthode d'Euler explicite, la méthode d'Euler implicite et la méthode de Störmer-Verlet/leapfrog.

Au sein de ce chapitre, nous allons mettre en évidence les sources de parallélisme que comporte une simulation de textiles : nous allons faire ressortir les calculs relatifs à une particule donnée qui sont totalement indépendants et ceux qui nécessitent des informations extérieures pour être réalisés. Pour mettre en évidence ces dépendances, des extraits d'algorithmes seront présentés. Nous tenons à signaler que ces algorithmes sont présentés sous leur forme la plus simplifiée (sans aucun soucis d'optimisation). Il est évident qu'ils ont été implanté d'une manière bien différente au sein de notre plate-forme afin de gagner en performance.

4.2 Boucle de la simulation

Une simulation de particules consiste à chaque pas de temps à calculer la position de ces particules. Dans le chapitre 2 section 2.5, les différentes étapes de la boucle de simulation ont été présentées. L'algorithme 3 les récapitule brièvement.

Algorithme 3 Boucle de la simulation de textiles

```
1 :   Répéter
2 :      Calcul_Force();
3 :      Calcul_Accel();
4 :      Integration();
5 :      Collision();
6 :   Fin
```

Notre objectif est de paralléliser cette boucle de simulation. La première stratégie consiste à paralléliser chacune de ces étapes sans se soucier des autres. En effet, le calcul des accélérations (ligne 3) nécessitent la connaissance des forces exercées sur chacune des particules ; celui des vitesses et des positions requièrent la connaissance des accélérations (ligne 4) ; et enfin le traitement des collisions consiste en une mise à jour de ces dernières (ligne 5). L'ordre des étapes ne peut donc être modifié. De plus, une étape donnée ne peut démarrer avant la fin de la précédente.

4.2.1 Calcul des forces

Dans le chapitre 2 section 2.2.3, les forces appliquées au temps t sur chacune des particules ont été présentées. Elles sont dues non seulement aux forces exercées par les ressorts (forces locales), mais également à des forces extérieures telles que la gravité

ou encore le vent (forces globales). Au sein de l'algorithme 4 nous pouvons voir les dépendances que comportent le calcul des forces exercées sur chacune des particules du système.

Algorithme 4 Calcul des forces exercées sur les particules

```
1 :   Fonction Calcul_Force(Force[], Position[], Vitesse[], Masse[], g, N) {
2 :   // Parametres : g le vecteur de gravite, N le nombre de particules
3 :       Pour i allant de 0 à N Faire
4 :           Force[i] = 0;
5 :           Pour j dans la liste des voisins de i Faire
6 :               // A_ij : calcule a partir de Position[i] et Position[j]
7 :               // B_ij : calcule a partir de Vitesse[i] et Vitesse[j]
8 :               Force[i] += (k_ij * A_ij) + (ν_ij * B_ij);
9 :           Fin de Pour
10 :          Force[i] += Masse[i] * g;
11 :      Fin de Pour
12 :  }
```

Par ailleurs, il faut noter que le calcul de l'interaction entre deux particules i et j n'est effectué en pratique qu'une seule fois sachant que $f_{ij} = -f_{ji}$.

En résumé, pour une particule donnée, le calcul des forces qui lui sont appliquées ne peut être réalisé sans la connaissance : (1) de sa position et (2) de sa vitesse ; (3) de sa masse ; (4) de son voisinage (particules voisines et caractéristiques des ressorts les reliant) ; (5) de la position et (6) de la vitesse de chacun de ses voisins. Le calcul des forces appliquées à une particule présente donc de fortes *dépendances par rapport aux particules voisines* qui sont en nombre restreint.

4.2.2 Calcul des accélérations

L'étape suivante dans la boucle de simulation (cf. algorithme 3) consiste à calculer les accélérations des particules. Dans le chapitre 2 section 2.2.3, nous avons vu que l'accélération d'une particule était donnée en appliquant la loi fondamentale de la dynamique élaborée par Newton. L'algorithme 5 présente ce calcul.

Algorithme 5 Calcul des accélérations des particules

```
1 :   Fonction Calcul_Accel(Accel[], Force[], Masse[], N) {
2 :   // Parametre : N le nombre de particules
3 :       Pour i allant de 0 à N Faire
4 :           Accel[i] = Force[i] / Masse[i];
5 :       Fin de Pour
6 :   }
```

Pour calculer l'accélération d'une particule donnée, il suffit donc de connaître sa masse et la force qui lui est appliquée. Ce calcul ne présente ainsi *aucune dépendance par rapport aux autres particules* contenues dans le système.

4.2.3 Schémas d'intégration explicites

L'intégration des accélérations des particules va permettre de connaître leurs vitesses ainsi que leurs positions. Dans le chapitre 3 sections 3.3.5 et 3.4.3, nous avons présenté en détails les schémas d'intégration explicites d'Euler et de Stöermer-Verlet/leapfrog. Mise à part une légère différence lors de leur démarrage (la méthode de leapfrog introduit un décalage de temps entre le calcul des vitesses et celui des positions), ces fonctions d'intégration s'implantent de la même manière. L'algorithme 6 présente cette implantation.

Algorithme 6 Intégration via un schéma explicite (Euler ou Stöermer-Verlet/leapfrog)

```
1 :   Fonction Integration(Position[], Vitesse[], Accel[], h, N) {
2 :   // Parametres : h le pas de temps, N le nombre de particules
3 :        Pour i allant de 0 à N Faire
4 :             Vitesse[i] = Vitesse[i] + h * Accel[i];
5 :             Position[i] = Position[i] + h * Vitesse[i];
6 :        Fin de Pour
7 :   }
```

Cet algorithme permet de montrer qu'en utilisant un schéma d'intégration explicite, le calcul de la vitesse d'une particule nécessite seulement la connaissance de son accélération et de sa vitesse au pas de temps précédent. De même le calcul de sa position requiert uniquement les valeurs de sa vitesse et de sa position précédente. Ces calculs sont ainsi *indépendants des autres particules*.

4.2.4 Schéma d'intégration implicite

L'intégration des accélérations peut également être effectuée en utilisant un schéma implicite. Les schémas implicites sont généralement plus stables que les schémas explicites mais sont hélas également plus compliqués à mettre en oeuvre du fait de la résolution d'un système d'équations non-linéaires à effectuer. La section 3.5.3 du chapitre 3 a présenté en détails le schéma d'Euler implicite. L'obtention des états des particules via cette méthode s'effectue notamment en un certain nombres d'étapes : (1) évaluer les matrices des dérivées des forces, (2) construire le système d'équations linéaires creux, (3) résoudre ce système, et enfin (4) mettre à jour les vitesses et les positions des particules.

Évaluation des matrices des dérivées des forces

Le schéma implicite employé dans notre simulation de textiles est basé sur la définition des matrices des dérivées des forces présentée par Volino et Magnenat-Thalmann (cf. chapitre 3, section 3.5.3). L'algorithme 7 présente le calcul des éléments diagonaux (lignes 9-11) et de ceux en dehors de la diagonale (lignes 6-8) des deux matrices des dérivées des forces.

Algorithme 7 Intégration via le schéma d'Euler implicite (étape 1)

```
1 :   Fonction Integration_etape1(Matrice_df_dx[][], Matrice_df_dv[][],
              Matrice_df_dx[], Matrice_df_dv[], Position[], Vitesse[], N) {
2 :   // Parametres : N le nombre de particules
3 :       Pour i allant de 0 à N Faire
4 :           Pour j dans la liste des voisins de i Faire
5 :               // C_ij : calcule a partir de Position[i] et Position[j]

6 :               // Elements en dehors de la diagonale
7 :               Matrice_df_dx[i][j] = k_ij * C_ij ;
8 :               Matrice_df_dv[i][j] = ν_ij * C_ij ;

9 :               // Elements de la diagonale
10 :               Matrice_df_dx[i] += Matrice_df_dx[i][j];
11 :               Matrice_df_dv[i] += Matrice_df_dv[i][j];

12 :           Fin de Pour
13 :       Fin de Pour
14 :   }
```

Cette étape dans l'algorithme d'intégration du schéma d'Euler implicite possède les mêmes contraintes de dépendances que le calcul des forces vu précédemment. En effet pour pouvoir calculer un élément donné de la matrice, il est nécessaire de connaître (1) la position de la particule considérée ; (2) son voisinage (particules voisines et propriétés physiques des ressorts les reliant) ; et enfin (3) la position de chacun de ses voisins. Le calcul d'un élément de la matrice présente donc de fortes *dépendances par rapport à son voisinage*.

Construction du système d'équations linéaires

L'étape suivante dans l'application du schéma d'Euler implicite pour l'intégration des accélérations consiste à construire le système linéaire de la forme $Ax = b$. Il faut donc remplir la matrice $A = \left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)$ et le vecteur $b = \left(hf(t) + h^2\frac{\partial f}{\partial x}v(t)\right)$. Pour cela il faut notamment effectuer la multiplication entre la matrice $\frac{\partial f}{\partial x}$ et le vecteur $v(t)$.

Cette matrice possède une structure creuse. En effet les seuls éléments non nuls j de la ligne i de la matrice correspondent aux voisins j de la particule i considérée. Mise à part l'élément de la diagonale, tous les autres éléments de cette ligne i sont nuls. La figure 4.1 présente la forme de la matrice $\frac{df}{dx}$ associée à un maillage triangulaire de 15 particules dans le plan 2D. Le nombre maximal de voisins pour chaque particule est alors de 6. Cela signifie que sur la ligne i de la matrice, correspondant à la particule i, il y a au plus 6 éléments non nuls en dehors de l'élément diagonal. L'élément diagonal représente la somme des contributions des forces dues à l'ensemble des voisins de i, alors que l'élément de la colonne j de la ligne i correspond seulement à la contribution du voisin j de la particule i. La matrice est alors tridiagonale et creuse.

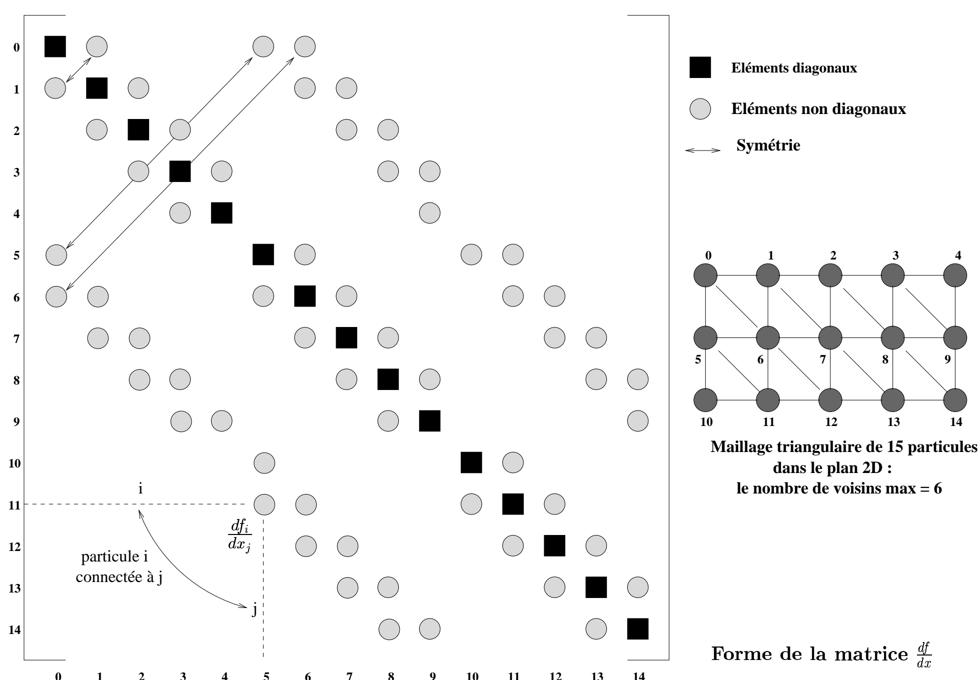


Figure 4.1 *Forme de la matrice $\frac{df}{dx}$ associé au maillage triangulaire des 15 particules dans le plan 2D : les éléments diagonaux sont non nuls et correspondent à $\frac{df_i}{dx_i}$, les éléments en dehors de la diagonale non nuls correspondent à $\frac{df_i}{dx_j}$ avec j voisin de la particule i sachant que $\frac{df_i}{dx_j} = \frac{df_j}{dx_i}$*

Lors du calcul du $i^{ème}$ élément du vecteur issu du produit entre la matrice $\frac{\partial f}{\partial x}$ et le vecteur $v(t)$, il est nécessaire de connaître tous les éléments de la ligne i de la matrice soient les contributions apportées par chacun des voisins j de l'élément i considéré. La construction du système linéaire fait donc également apparaître des *dépendances dues aux voisinages des particules*.

Résolution par la méthode du Gradient Conjugué

L'avant dernière étape du schéma d'Euler implicite consiste en la résolution du système linéaire creux construit précédemment. Pour cela nous employons la méthode du Gradient Conjugué détaillé dans le chapitre A de l'annexe. Nous rappelons ici son algorithme :

Algorithme 8 Intégration via le schéma d'Euler implicite (étape 3)

1 : $\beta \leftarrow 0;$ `// Initialisation du facteur d'erreur`
2 : $x \leftarrow 0;$ `// Initialisation de la solution`
3 : $r \leftarrow b - Ax;$ `// Initialisation du vecteur de residu`

4 : **Faire**
5 : $\alpha \leftarrow r^T r;$ `// Initialisation du pas`

6 : **Si** $(\beta \neq 0)$
7 : $d \leftarrow r + (\frac{\alpha}{\beta})d;$ `// Calcul de la nouvelle direction`
8 : **Sinon**
9 : $d \leftarrow r;$ `// Initialisation du vecteur de direction`

10 : $\beta \leftarrow d^T Ad;$ `// Calcul du facteur d'erreur`
11 : $r \leftarrow r - (\frac{\alpha}{\beta})Ad;$ `// Calcul du vecteur de residu`
12 : $x \leftarrow x + (\frac{\alpha}{\beta})d;$ `// Calcul de la solution`
13 : $\beta \leftarrow \alpha;$ `// Nouveau facteur d erreur`

14 : **Jusqu'a** $(\beta < \varepsilon)$ `// Iteration jusqu'a la precision souhaitee`

La matrice A présente dans cet algorithme correspond à la matrice calculée dans l'étape précédente. Étant construite à partir de produits scalaires impliquant les matrices des contributions des forces, elle possède également une structure creuse. Cette matrice doit être multipliée à chaque itération de l'algorithme par un vecteur de directions d (lignes 10-11). Les mêmes contraintes de *dépendances* que celles vues précédemment sont donc présentes dans cet algorithme.

Nous pouvons par ailleurs noter que cet algorithme est itératif est donc comporte une boucle dont les itérations s'arrêtent quand la précision souhaitée est atteinte. La simulation comporte donc des étapes ayant des pas de temps qui peuvent être différents de celui de la boucle principale de la simulation. Il sera sans doute intéressant de “jouer” avec ces différents pas de temps afin de trouver le meilleur compromis entre la précision recherchée et le temps d'exécution.

Mise à jour des positions et vitesses

Après avoir résolu le système linéaire, la dernière étape de l'application du schéma d'Euler implicite consiste à mettre à jour les position et vitesse de chacune des particules. L'algorithme 9 présente cette dernière étape.

Algorithme 9 Intégration via le schéma d'Euler implicite (étape 4)

```
1 :  Fonction Integration(Position[], Vitesse[], X[], h, N) {
2 :  // Parametres : h le pas de temps, N le nombre de particules
3 :      Pour i allant de 0 à N Faire
4 :          Vitesse[i] = Vitesse[i] + X[i] ;
5 :          Position[i] = Position[i] + h * Vitesse[i] ;
6 :      Fin de Pour
7 :  }
```

La mise à jour de la vitesse de la $i^{ème}$ particule nécessite donc la connaissance de sa vitesse au pas de temps précédent, ainsi que du $i^{ème}$ élément du vecteur x solution du système linéaire $Ax = b$. Pour la mise à jour de la position de cette particule, il suffit de connaître sa vitesse et sa position précédente. Ces mises à jours ne présentent donc *aucune dépendance par rapport aux autres particules* contenues dans le système.

4.3 Conclusion

A chaque pas de temps d'une simulation de textiles il faut tout d'abord calculer les forces exercées sur chacune des particules, puis en déduire leurs accélérations et enfin appliquer un schéma d'intégration permettant l'obtention des nouvelles vitesses et positions des particules.

Les forces appliquées sur une particule donnée ne peuvent être calculées que si le *voisinage de la particule* est connu. Il est en effet indispensable de connaître pour ce calcul les particules qui sont reliées à la particule considérée. Ces liaisons sont effectuées via des ressorts dont les caractéristiques physiques de raideur et d'amortissement doivent également être connues.

Lors de l'utilisation d'un schéma d'intégration explicite, tels que le schéma d'Euler explicite ou encore celui de Störmer-Verlet/leapfrog, aucune connaissance des particules voisines n'est nécessaire pour évaluer ensuite tous les autres calculs relatifs à une particule donnée (accélération, position, vitesse).

Dans le cas d'un schéma implicite tel que celui d'Euler implicite (employé pour des raisons de stabilité par exemple), les dépendances de données dues au voisinage des particules réapparaissent lors de la construction et la résolution du système linéaire qui permet l'obtention des nouvelles vitesses et positions des particules.

Ces dépendances de données engendrent des difficultés lors de la parallélisation de la boucle de simulation. En effet, pour effectuer certains calculs, tels que les forces ou encore ceux relatifs à la méthode d'Euler implicite, les processus devront accéder à des espaces de données disjoints si les données nécessaires au calcul sont placées sur des processeurs différents. C'est pourquoi il faudra faire attention à la *localité des données et des calculs* afin d'éviter d'avoir à transférer trop de données d'un processeur à un autre.

Il y aura donc un compromis à faire entre choisir une méthode d'intégration simple à implanter en parallèle mais dont l'instabilité numérique limite la grandeur de son pas de temps ; ou choisir une méthode permettant d'opter pour de larges pas de temps mais qui est plus difficile à implanter en parallèle car chacun des calculs relatifs à une particule nécessite des informations sur les particules voisines et sur les liaisons entre ces particules. Le tableau 4.1 récapitule ces critères de sélection liés aux différentes méthode d'intégration.

	Méthodes d'intégration		
	Euler explicite	*Verlet*	*Euler implicite*
Stabilité	Instable (p. 35)	Stable (p. 38)	Incond. stable (p. 40)
Pas de temps	Faible (p. 35)	Faible (p. 38)	Large (p. 40)
Parallélisation	Simple	Simple	Difficile
Tâches indépendantes	Intégration (p. 50) Accélération (p. 49)	Intégration (p. 50) Accélération (p. 49)	Accélération (p. 49)
Tâches localement dépendantes	Force (p. 48)	Force (p. 48)	Force (p. 48) Intégration (p. 50)

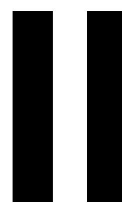
TAB. 4.1 *Tableau récapitulatif sur les méthodes d'intégration implantées (Euler explicite, Störmer-Verlet/leapfrog, Euler implicite) : stabilité, pas de temps, interactions des tâches de calculs, facilité de parallélisation. Les numéros de page relatifs aux explications étant indiqués entre parenthèses.*

Il est également important de noter que ces *dépendances sont faibles* car chacune des particules ne possède qu'un nombre limité de voisins. Ce faible voisinage engendre notamment des *structures de données creuses* dont il faudra tenir compte lors de l'implantation.

D'autre part, l'emploi de la méthode d'Euler implicite nécessite la résolution d'un système linéaire effectuée par la méthode itérative du Gradient Conjugué. À l'intérieur de la boucle principale de la simulation, il peut donc y avoir d'autres boucles qui ont des pas de temps différents. Nous pouvons alors introduire une notion de *coût d'une seconde de simulation*, noté C_s. Nous avons comme relation $C_s = C_h/h$, avec h le pas de temps de la simulation et C_h son coût. À coût constant nous pouvons soit opter pour une méthode

d'intégration impliquant un C_h important mais avec un pas de temps important (Euler implicite) ; ou pour une méthode générant un faible C_h avec des pas de temps petits (méthodes explicites).

En résumé les caractéristiques essentielles d'une simulation de textiles sont :
- une boucle infinie avec un pas de temps h,
- des sous-boucles pouvant avoir différents pas de temps,
- la présence de calculs totalement indépendants et d'autres présentant des dépendances locales de données,
- des dépendances de données correspondant au voisinage d'une particule,
- la présence de structures de données creuses.



Parallélisme

L'objectif de cette thèse concerne l'élaboration d'algorithmes parallèles destinés à la synthèse d'image et plus particulièrement à l'animation de textiles. Une simulation de tissu se présente sous la forme d'une boucle infinie dans laquelle, à chaque pas de temps, doivent être calculés les états de particules décrivant l'objet déformable. Les mêmes calculs sont donc effectués sur chacune des particules. Certains calculs sont totalement indépendants les uns des autres, c'est-à-dire qu'ils n'utilisent aucune information relative aux autres particules. Par contre d'autres nécessitent des informations précises sur le voisinage de la particule considérée. Ensuite à partir des résultats issus de la simulation, la visualisation de l'objet déformable doit être effectuée. Un couplage entre la simulation parallèle et le rendu doit être ainsi établi, sachant que le calcul du rendu peut également être parallélisé.

Cette application de simulation de textiles doit être à terme exécutée sur une grappe de machines. Comment élaborer ces algorithmes parallèles de manière efficace ? Quel est l'intérêt d'utiliser une grappe de PCs plutôt qu'une machine parallèle traditionnelle ? Comment programmer ces algorithmes pour les exécuter sur la grappe de PCs ? Et enfin comment établir un couplage entre deux programmes parallèles ? Voici l'ensemble des questions auxquelles cette partie va tenter de répondre.

Par ailleurs, cette deuxième partie se termine par la présentation de l'environnement de programmation parallèle ATHAPASCAN. *Cet environnement a en effet faciliter la parallélisation de notre application.*

Calcul parallèle 5

L'un des objectifs de cette thèse est l'élaboration d'algorithmes parallèles permettant d'effectuer une simulation de textiles sur une grappe de PCs. Avant de voir la parallélisation propre à notre application, il est important de montrer au lecteur comment s'élaborent de tels algorithmes. Ce chapitre présente donc les différentes étapes à suivre pour transformer les spécifications d'un problème initial en un algorithme parallèle. Nous verrons également quels sont les critères qui permettent d'évaluer cet algorithme parallèle.

5.1 Introduction

Dans différents domaines tels que la physique, la biologie, l'océanographie, la météorologie ou encore l'astronomie, les scientifiques cherchent à simuler des phénomènes naturels de plus en plus complexes afin de mieux les comprendre. Prenons l'exemple de la météorologie. Il n'est pas évident de prédire le temps qu'il fera demain ou encore si une tempête risque de s'abattre sur une ville. Pour effectuer ces prédictions, les météorologues effectuent des simulations. Ces simulations sont élaborées à partir de modèles mathématiques, basés à la fois sur des observations physiques des mouvements des différents constituants de l'atmosphère mais également à partir d'une grande masse de données accumulées au fil des années. Ces simulations requièrent donc à la fois une grande puissance de calculs ainsi qu'une grande place mémoire nécessaire au stockage de plusieurs Giga-Octets de données.

En informatique graphique, suite à un engouement général pour les jeux vidéos, les films d'animation [1] ou encore la réalité virtuelle, l'animation d'objets 3D a énormément évolué vers des algorithmes simulant des comportements de plus en plus réalistes mais également de plus en plus complexes. Ces animations nécessitent désormais une puis-

sance de calcul importante aussi bien pour leur simulation que pour la visualisation de leurs résultats.

Les architectures parallèles offrent une puissance de calcul et une capacité de stockage potentiellement très importantes. Certains problèmes complexes jusqu'alors laissés en suspens faute de puissance nécessaire ou de mémoire suffisamment importante à leur résolution peuvent alors être reconsidérés. Mais la difficulté réside dans l'exploitation de toutes les ressources disponibles sur ce type d'architecture.

C'est pourquoi ce chapitre présente tout d'abord la méthodologie permettant l'élaboration de l'algorithme parallèle relatif à notre problème initial. Puis les différents critères permettant d'évaluer les performances des algorithmes parallèles seront présentés. Le lecteur pourra noter que ce chapitre s'inspire très largement du livre de Ian Foster [44] intitulé “Designing and building parallel programs”.

5.2 Conception d'algorithmes parallèles

Dans la partie précédente, le problème relatif à la simulation de textiles a été détaillé en faisant apparaître les sources potentielles de difficulté en ce qui concerne sa parallélisation. En effet certains calculs présentent de fortes dépendances de données entraînant une attention particulière sur la localité des données. Dans cette section nous allons étudier comment la spécification d'un problème initial peut être transformée en un algorithme parallèle [19, 66, 103, 44, 75, 12]. Pour cela, il est nécessaire de faire apparaître les concurrences liées au problème et de faire attention non seulement à la localité des données mais également à la notion de passage à l'échelle de l'algorithme parallèle créé.

5.2.1 Méthodologie

Il y a cinq étapes illustrées par la figure 5.1, pour passer de la spécification du problème initiale à l'élaboration de l'algorithme parallèle [44] :

1. *Partitionnement.* Le problème initial est décomposé en tâches de calculs plus petites. Cette décomposition est effectuée sans se soucier du nombre de processeurs sur lequel le calcul sera ensuite exécuté, mais l'attention est plutôt orientée sur les opportunités apportées par une exécution parallèle.
2. *Communication.* Les communications nécessaires entre les tâches sont mises en évidence permettant de définir les structures algorithmiques et de communication appropriées.
3. *Agglomération.* Les tâches et les structures de communication définies précédemment sont évaluées et si nécessaire des tâches sont combinées en tâches plus grosses afin d'améliorer les performances et de réduire les coûts.

4. *Ordonnancement.* Chaque tâche est assignée à un processeur de façon à optimiser l'utilisation des processeurs et à minimiser les coûts des communications. Le placement peut être spécifié de manière statique ou déterminé lors de l'exécution en utilisant des algorithmes d'équilibrage de charge.
5. *Interaction.* Les interactions entre la simulation et la visualisation doivent être gérées dans le cadre d'une simulation graphique.

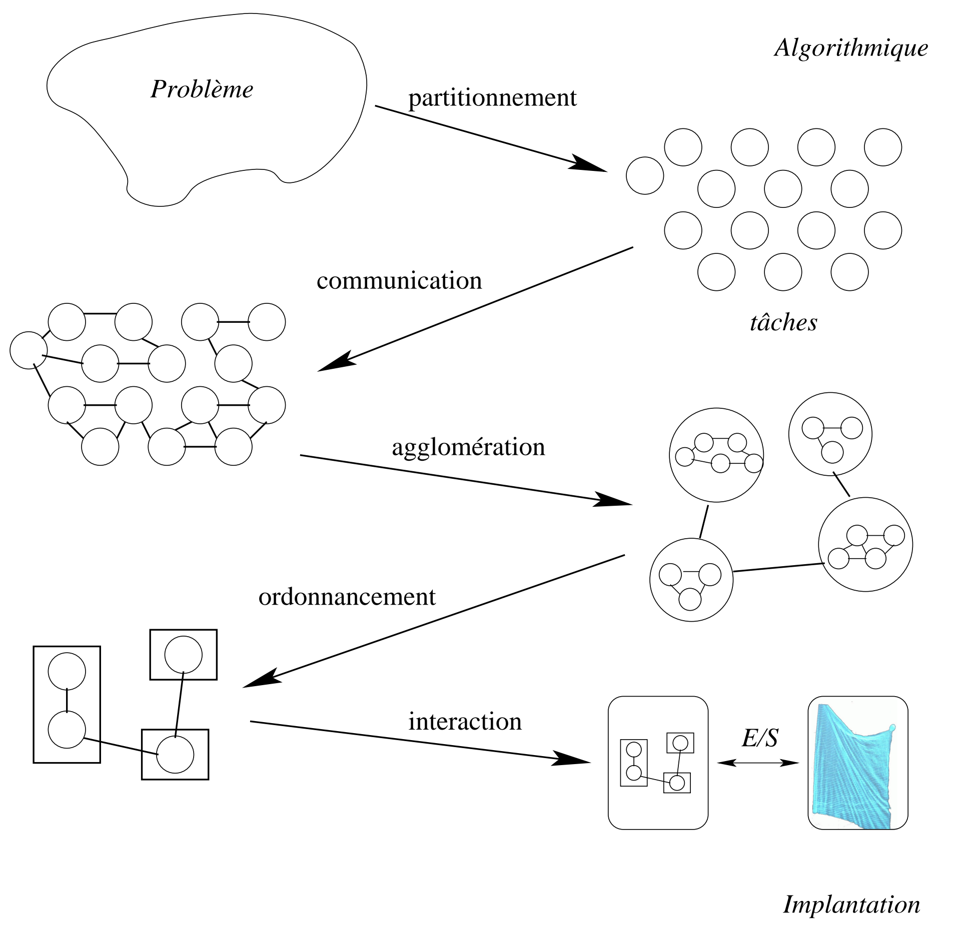


Figure 5.1 *Méthodologie pour la conception de programmes parallèles. Le problème initial est partitionné en sous problèmes ; les communications nécessaires sont établies ; les sous problèmes sont ensuite agglomérés en tâches ; ces tâches sont ensuite placées sur les processeurs ; enfin les interactions entre la simulation et la visualisation sont gérées.*

Nous allons à présent détailler ces cinq étapes fondamentales dans l'élaboration d'un algorithme parallèle.

5.2.2 Partitionnement

L'objectif du partitionnement consiste à tirer le plus grand profit d'une exécution parallèle. Il faut alors identifier toutes les tâches qui peuvent être exécutées en concurrence. Nous obtenons alors une décomposition du problème avec un grain très fin c'est-à-dire que le problème initial a été fragmenté en un grand nombre de tâches de taille assez petite. Lors de l'évaluation des communications qui seront nécessaires entre ces tâches, cette décomposition sera modifiée en agglomérant des tâches ensemble afin d'en obtenir de nouvelles de taille plus grande permettant de réduire le coût en terme de communication.

Le partitionnement décompose en taille plus réduite à la fois les calculs associés à un problème que l'ensemble des données sur lesquelles les calculs seront effectués. Deux techniques de partitionnement peuvent alors être différenciées. La première consiste à tout d'abord étudier les données nécessaires au problème, puis à chercher la décomposition de ces données la plus adaptée pour ensuite identifier les calculs qui leur seront appliqués. Nous parlons alors de *décomposition de l'espace*. L'autre approche consiste à commencer par décomposer les calculs et ensuite de travailler sur les données. Nous parlons alors de *décomposition fonctionnelle*. Ce sont deux techniques complémentaires que nous avons appliquées ensemble lors de la parallélisation de notre simulation de textiles.

Dans cette première étape dans l'élaboration de l'algorithme parallèle associé à notre problème, il est nécessaire d'éviter toute redondance de calculs et toute réplication de données. Le but est de rechercher des tâches permettant de partitionner à la fois les calculs et les données en ensembles distincts. Dans une étape ultérieure les possibilités de réplication de données ou de calculs pourront être envisagées afin de réduire les communications.

Décomposition de l'espace

La technique de décomposition de l'espace permettant de partitionner le problème initial, débute par la recherche d'une décomposition des données. Les données sont fragmentées en morceaux de taille plus petite et si possible de même taille. Ensuite les calculs relatifs au problème sont à leur tour décomposés. Généralement ils sont établis de façon à traiter les fragments de données obtenus précédemment. Dans ce cas leur temps d'exécution est lié à la taille des fragments de données qu'ils ont à traiter. Ce partitionnement permet alors l'obtention d'un ensemble de tâches. Chacune de ces tâches comporte à la fois des fragments de données et un ensemble d'opérations permettant leur traitement. Une opération peut nécessiter des données de différentes tâches. Dans ce cas des communications sont requises pour faire transiter les données entre ces tâches. Tout l'enjeu réside dans la recherche de la structure de données la mieux adaptée à cette décomposition des données. Il faut alors bien analyser quelles sont les données qui sont le plus souvent requises.

La figure 5.2 illustre la technique de décomposition de l'espace dans un problème

impliquant une grille à trois dimensions pouvant représenter l'espace 3D dans le cadre de notre simulation. Chacun des points de la grille 3D peut représenter une particule du tissu. Le même calcul, par exemple celui des positions, doit donc être effectué en chacun de ces points. Des décompositions selon les axes x, y et/ou z sont alors possibles. Dans cette première étape nous privilégions la décomposition impliquant le plus grand nombre de tâches, c'est-à-dire qu'ici une tâche sera définie pour chacun des points de la grille à traiter, c'est-à-dire qu'elle ne traitera qu'une seule particule de la simulation.

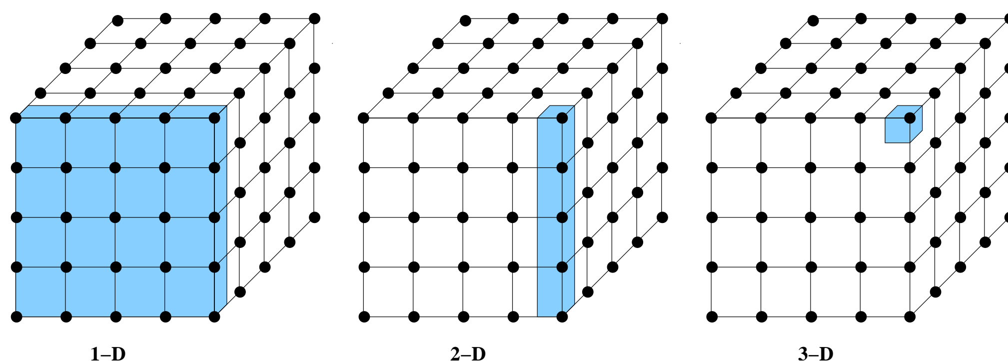


Figure 5.2 *Décomposition de l'espace pour un problème impliquant une grille à trois dimensions. Des décompositions à une-, deux- ou trois-dimensions sont possibles. Dans la majorité des cas, la décomposition en trois dimensions est privilégiée, offrant le plus de flexibilité.*

Décomposition fonctionnelle

Lors d'une décomposition fonctionnelle, l'analyse se porte essentiellement sur les calculs et non plus sur les données comme dans le cadre de la décomposition de l'espace. Ensuite, après avoir réussi à diviser le problème en plusieurs tâches disjointes, les données nécessaires à ces tâches sont étudiées. Ces données peuvent être disjointes et dans ce cas la décomposition est terminée. Dans le cas contraire, si les données se chevauchent considérablement, c'est-à-dire si plusieurs tâches requièrent les mêmes données, les communications nécessaires à ces calculs risquent d'être trop importantes afin d'éviter la réplication de données. Il vaut mieux alors reconsidérer la décomposition en appliquant une décomposition de l'espace.

Cette méthode de décomposition est généralement utilisée dans le but de découper un problème complexe en plusieurs sous-modules plus ou moins indépendants, qui sont ensuite traités séparément. Dans notre cas, la simulation de textiles se décompose en plusieurs sous-modules correspondants aux différents calculs à effectuer sur chacune des particules : les forces, les accélérations, les vitesses et les positions. Si nous supposons qu'une tâche de calcul ne s'applique qu'à une seule particule, nous avons vu que certaines

d'entres elles nécessitent des informations sur leurs particules voisines (forces, intégration implicite), alors que d'autres peuvent être traitées indépendamment (accélération, intégration explicite). Deux types de taches de calculs sont ainsi présentes au sein de la simulation : des tâches indépendantes et des tâches dépendantes par rapport aux données contenues dans d'autres tâches.

Propriétés d'un partitionnement efficace

Ian Foster [44] présente une liste de points à vérifier après avoir partitionné son problème en différentes tâches pour savoir si la décomposition a été effectuée judicieusement. Il y a quatre points majeurs à vérifier :

1. Le partitionnement du problème initial doit comporter plus de tâches que de processeurs disponibles sur la machine cible afin d'être le plus efficace possible lors de l'exécution parallèle. Dans le cas contraire, des processeurs se retrouveraient sans tâche à exécuter. Mais il faut faire attention à ne pas créer de trop petites tâches sous prétexte de faire travailler tous les processeurs, car elles impliqueraient sans doute un grand nombre de communications et la parallélisation serait dès lors peu efficace. Il y a donc un compromis à trouver entre la taille des tâches, leur nombre, la quantité de communication engendrée, ainsi que le nombre de processeurs à utiliser.
2. Afin d'obtenir un programme efficace sur un grand nombre de machines, le partitionnement doit être effectué en évitant les redondances de calcul et de stockage de données.
3. Pour faciliter l'équilibrage de charges entre les processeurs, les tâches doivent être de préférence de même taille.
4. Il est préférable que l'augmentation de la taille du problème initial fasse augmenter le nombre de tâches plutôt que la taille des tâches. Dans le cas contraire l'algorithme parallèle ne serait pas capable de résoudre des problèmes de tailles plus grandes même avec un nombre plus important de processeurs disponibles.

5.2.3 Communication

Les tâches résultant d'une décomposition ont été établies de façon à ce qu'elles soient exécutées en concurrence, mais elles ne peuvent généralement être totalement indépendantes les unes des autres. Les calculs devant être exécutés au sein d'une tâche nécessitent généralement des données présentes dans d'autres tâches. Les données doivent alors être transférées entre les tâches afin de permettre l'exécution des calculs. Ces flux d'information sont spécifiés dans la phase de communication de l'élaboration de l'algorithme parallèle.

Les communications au sein de notre application parallèle seront gérées par l'environnement de programmation parallèle ATHAPASCAN brièvement présenté dans le chapitre

suivant. Quelques points sont tout de même à vérifier avant de passer à l'étape suivante dans l'élaboration de l'algorithme parallèle, afin de ne pas effectuer de communications inutiles ou qui auraient pu être évitées :

1. Il est conseillé d'obtenir à la fin de la décomposition, un ensemble de tâches qui requièrent approximativement le même volume de communication. Dans le cas contraire, le fait de ne pas avoir réussi à équilibrer les communications entre les tâches risque de rendre difficile le passage à l'échelle de l'algorithme parallèle. Il vaut alors mieux essayer de remanier les tâches. Par exemple, si une structure de données encapsulée dans une tâche est fréquemment accédée par d'autres tâches, il est conseillé de penser à distribuer ou à répliquer cette structure de données, permettant ainsi de réduire les communications qu'elle engendrait.
2. Si possible, il faut réussir à créer des tâches de calculs de façon à ce qu'elles n'aient à communiquer qu'avec un ensemble réduit de voisins.
3. Les opérations de communications doivent pouvoir être effectuées en concurrence afin d'obtenir un algorithme efficace et passant à l'échelle.
4. Les calculs associés aux différentes tâches doivent également pouvoir être exécutés en concurrence afin d'obtenir un algorithme efficace et passant à l'échelle. Il est recommandé d'essayer de recouvrir les communications par des calculs en réorganisant leur ordre initial d'exécution.

5.2.4 Agglomération

Les deux premières étapes de l'élaboration de l'algorithme parallèle ont permis l'obtention d'un partitionnement des calculs en un ensemble de tâches et l'établissement des communications nécessaires entre ces tâches afin de fournir les données indispensables à leur exécution. L'algorithme qui en résulte n'est pour le moment pas des plus efficaces car il crée sans doute beaucoup plus de tâches que de processeurs disponibles sur l'architecture parallèle visée. De plus l'exécution de petites tâches n'améliore généralement pas l'efficacité d'une exécution parallèle.

La troisième étape consiste donc à remanier le partitionnement et les phases de communication en vue d'obtenir un algorithme efficace sur différents types de machines parallèles. L'étude porte sur la possibilité d'agglomérer ensemble plusieurs tâches afin d'en diminuer le nombre et d'augmenter leur taille. Les possibilités de réplication de données et/ou de calculs sont également envisagées lors de cette étape d'agglomération.

La figure 5.3 reprend le cas du problème impliquant une grille à 3 dimensions avec une décomposition 3D de l'espace de simulation. Afin d'obtenir des tâches de taille plus importante que lors de la première décomposition, des tâches adjacentes sont combinées permettant d'augmenter la granularité du partitionnement. Chacune des tâches est alors chargée de traiter non plus une seule particule de la simulation mais un ensemble de particules. La granularité des tâches correspond alors aux nombres de particules qu'elles ont

à traiter. Le paragraphe suivant étudie cette solution consistant à augmenter la granularité afin d'obtenir une décomposition plus efficace en terme d'exécution parallèle.

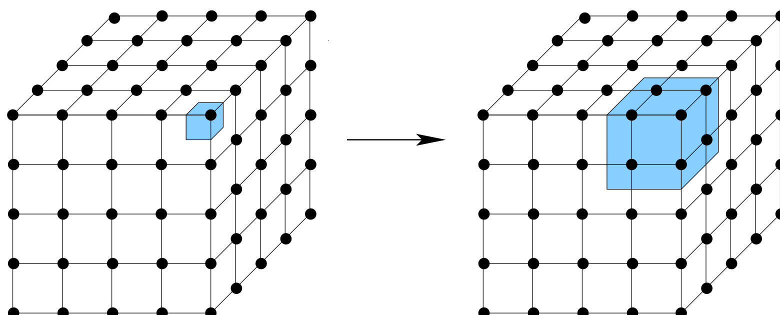


Figure 5.3 *Décomposition de l'espace pour un problème impliquant une grille à 3 dimensions. Une décomposition en trois dimensions a été privilégiée. Des tâches adjacentes ont été combinées permettant l'augmentation de la granularité initialement obtenue.*

Augmentation de la granularité

Le fait d'avoir défini un très grand nombre de tâches à grain fin lors des premières étapes de l'élaboration de l'algorithme parallèle limite l'efficacité de son exécution sur une architecture parallèle. Le point le plus critique d'une telle exécution réside dans le coût en communication. En effet avec certains types de modèles de programmation parallèle, les calculs sont stoppés lors de la réception ou de l'envoi de messages. La performance de l'algorithme peut donc être considérablement acrue si le temps passé à communiquer est diminué ou recouvert. Cette amélioration peut donc être obtenue en diminuant la quantité de messages envoyés. Ce résultat peut également être obtenu en utilisant moins de messages tout en préservant la même quantité de données transitées. En effet, le coût d'une communication comprend un coût fixe et n'est pas simplement proportionnel à la quantité de données envoyées.

En plus du coût de communication, il ne faut pas oublier qu'il existe un coût de création de tâches qui fait partie dans notre cas du surcoût lié à l'utilisation d'ATHAPASCAN. Le fait de diminuer le nombre de tâches va donc également permettre de diminuer ce surcoût. Les figures 5.4 et 5.5 illustrent les effets de l'augmentation de la granularité sur le coût de communication.

Sur la figure 5.4 le calcul sur une grille 8×8 a été partitionné en $8 \times 8 = 64$ tâches, chacune étant responsable d'un seul point. Tandis que dans la figure 5.5 le même calcul a été partitionné en $2 \times 2 = 4$ tâches, chacune étant responsable de 16 points. Dans le premier cas, le coût en communication pour chacune des tâches est de 4, soit en coût total de $64 \times 4 = 256$ communications, soit un transfert de 256 données. Dans le second cas, seulement 4×4 communications sont nécessaires soit $16 \times 4 = 64$ données transférées.

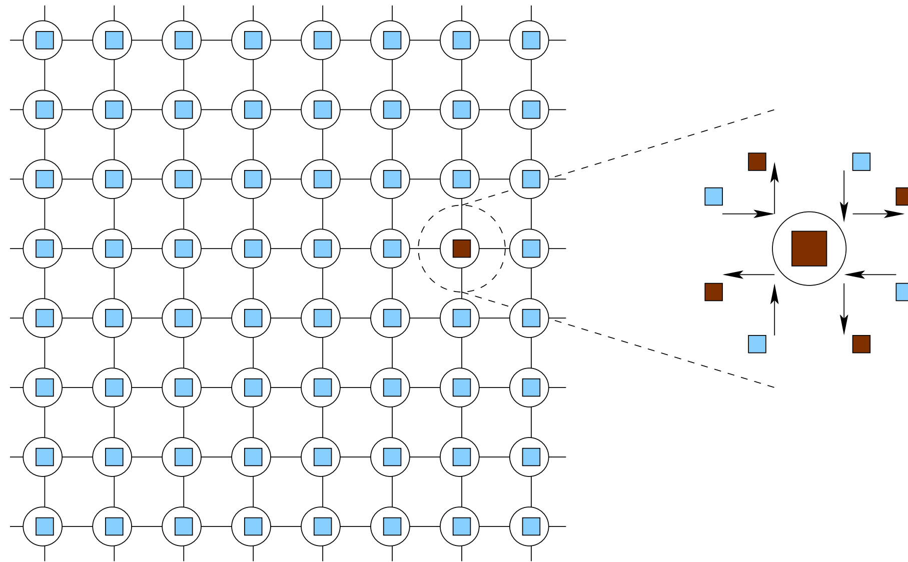


Figure 5.4 *Partitionnement 2D fin d'un calcul établi en chacun des points d'une grille* 8×8 *avec* 64 *tâches. Les communications entrantes (claires) et sortantes (foncées) d'une tâche donnée sont détaillées.*

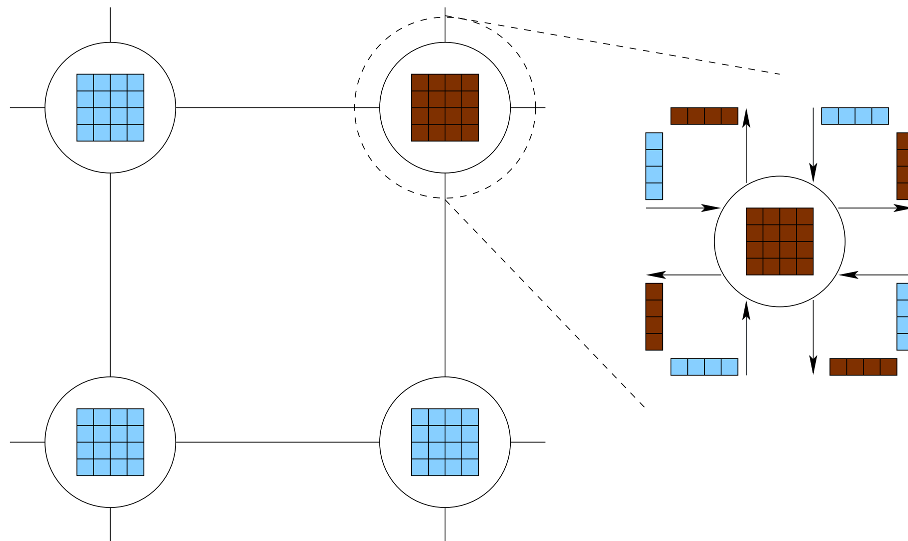


Figure 5.5 *Partitionnement 2D grossier d'un calcul établi en chacun des points d'une grille* 8×8 *avec* 4 *tâches. Les communications entrantes (claires) et sortantes (foncées) d'une tâche donnée sont détaillées.*

Les figures 5.4 et 5.5 permettent d'illustrer le fait que dans le cas où le volume de communication par tâche est petit, il est possible de réduire le nombre de communications et le volume total de communications en augmentant la granularité du partitionnement. Pour cela, plusieurs tâches sont agglomérées ensemble pour n'en obtenir plus qu'une seule. En effet, les communications requises par une tâche sont proportionnelles à la surface du sous-espace où la tâche opère, alors que les calculs nécessaires sont proportionnels au volume du sous-espace. La quantité de communications nécessaires à une unité de calcul (le rapport communication/calcul) diminue donc quand la taille de la tâche augmente.

Par conséquence, il est généralement plus efficace d'effectuer un partitionnement de dimension maximale, permettant de réduire l'aire de la surface (communication) nécessaire pour un volume donné (calcul). C'est pourquoi du point de vue de l'efficacité, il est préférable d'augmenter la granularité en agglomérant des tâches dans toutes les dimensions plutôt que de réduire la dimension du partitionnement.

5.2.5 Ordonnancement

Cette avant dernière étape de l'élaboration de l'algorithme parallèle sert à spécifier sur quels processeurs et à quels moments seront exécutées les différentes tâches créées lors du partitionnement du problème initial. Cette étape dépend fortement de l'architecture parallèle utilisée. L'objectif principal des algorithmes de placement est de diminuer le temps d'exécution. Deux règles sont employées pour y arriver : (1) les tâches pouvant être exécutées en concurrence sont placées sur des processeurs différents ; (2) les tâches communiquant fréquemment ensemble sont également placées sur le même processeur.

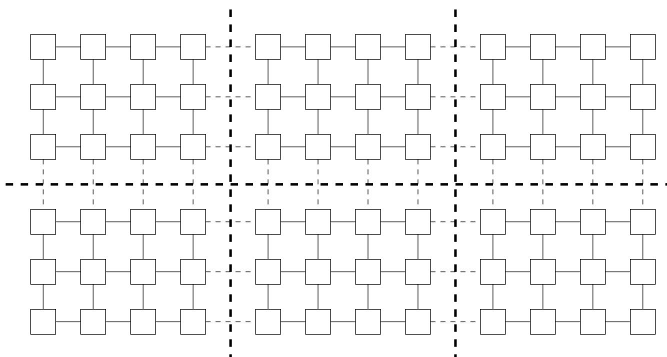


Figure 5.6 *Placement d'un problème de calcul basé sur une grille. Chaque tâche effectue la même quantité de calcul et ne communique qu'avec ces quatre voisins. Les lignes épaisses en pointillés délimitent les frontières des processeurs.*

Ces deux règles peuvent rentrer en conflit. De plus, les ressources matérielles limitent souvent le nombre de tâches pouvant être placées sur le même processeur. Il faut noter que

les problèmes de placement sont connus pour être des problèmes $\mathcal{NP}$-complets, c’est-à-dire qu’ils n’admettent pas de solution algorithmique en temps polynomial. Mais l’emploi d’heuristiques permet souvent d’améliorer ces algorithmes.

Les algorithmes utilisés pour le placement des tâches créées à partir d’une décomposition de l’espace, s’arrangent pour obtenir un nombre fixe de tâches de même taille et essayent de structurer les communications locales et globales. Un placement efficace est alors obtenu en minimisant les communications entre les processeurs. La figure 5.6 permet d’illustrer ce cas avec le placement des tâches associées à un problème de calcul basé sur une grille. La grille est partitionnée de façon à fournir la même quantité de calcul pour chacun des processeurs et à diminuer les communications entre les processeurs. En effet chacune des tâches effectue la même quantité de calcul et ne communique qu’avec ses quatre voisins.

Nous verrons que l’environnement de programmation parallèle ATHAPASCAN fournit un certain nombre d’algorithmes de placement avec la possibilité d’intégrer son propre algorithme d’ordonnancement.

5.2.6 Interaction

La spécificité d’un algorithme parallèle de simulation graphique par rapport à un algorithme parallèle “normal” réside dans le couplage à effectuer entre la partie simulation et la partie visualisation de l’application. Ces deux parties étant parallélisées, ce couplage revient à coupler différents programmes parallèles, ces programmes pouvant s’exécuter sur la même machine parallèle ou sur des machines distinctes.

Il est alors nécessaire de concevoir des méthodes permettant un partage optimal des ressources entre les programmes parallèles et optimisant la gestion des entrées/sorties entre ces programmes. De plus, pour obtenir des animations temps réel, il faut également gérer le fait que ces programmes parallèles peuvent avoir des vitesses d’exécution différentes (fréquence d’affichage, pas de temps de l’intégration de la simulation numérique, pas de temps du rendu, ...).

De nombreuses recherches relatent du problème posé par la gestion des entrées/sorties parallèles. Par exemple, Vilayannur *et al.* [123] présentent une stratégie basée sur l’utilisation de caches au sein même d’applications. Nous invitons le lecteur à se référer à l’article de Thakur *et al.* [119] présentant un résumé des diverses techniques employées dans ce domaine avec notamment la description de MPI I/O.

Le couplage de notre animation de textiles est étudié en détails au sein du chapitre 11.

5.3 Modèles de performance

Nous allons à présent introduire les principaux critères d'évaluation pour mesurer les performances des programmes parallèles.

Un algorithme séquentiel est généralement évalué en terme de temps d'exécution exprimé en fonction de la taille de ses entrées. Le temps d'exécution d'un algorithme parallèle dépend non seulement de la taille de ses entrées mais également de l'architecture parallèle sur laquelle il est exécuté et du nombre de processeurs disponibles. C'est pourquoi un algorithme parallèle ne peut être évalué sans tenir compte de l'architecture parallèle. Un système parallèle est donc la combinaison d'un algorithme et d'une architecture parallèle sur laquelle il est implanté.

5.3.1 Temps d'exécution

Le *temps d'exécution séquentiel* d'un programme correspond au temps écoulé entre le début et la fin de son exécution sur une machine séquentielle. Le *temps d'exécution parallèle* est le temps écoulé entre le moment où le calcul parallèle débute et le moment où le dernier processeur termine son exécution. Notons T_{seq} le temps d'exécution séquentiel et T_P le temps d'exécution parallèle.

5.3.2 Accélération

Lors de l'évaluation d'un système parallèle, il est intéressant de connaître le gain de performance obtenu en parallélisant une application donnée par rapport à son implantation séquentielle. L'*accélération* permet de mesurer ce gain. Elle est définie comme le rapport entre le temps mis pour résoudre le problème sur un unique processeur et le temps mis pour résoudre ce même problème sur une machine parallèle avec p processeurs identiques. L'accélération est notée A_P et est donc formulée de la façon suivante :

$$A_P = \frac{T_{seq}}{T_P}.$$

Il existe généralement plusieurs algorithmes séquentiels permettant de résoudre un problème donné. Le temps séquentiel employé dans le calcul de l'accélération correspond au temps du programme séquentiel le plus rapide permettant la résolution du problème considéré. De plus les p processeurs employés lors de l'exécution parallèle sont supposés identiques à celui utilisé lors de l'exécution séquentielle.

5.3.3 Efficacité

Seul un système parallèle idéal contenant p processeurs peut permettre l'obtention d'une accélération de p. En pratique le comportement idéal n'est jamais atteint car durant

l'exécution parallèle les processeurs ne passent pas 100% de leur temps dans les calculs de l'algorithme.

L'*efficacité* permet donc de mesurer le temps effectivement passé par les processeurs dans l'algorithme. Elle est définie comme le rapport entre l'accélération et le nombre de processeurs. Dans le cas d'un système parallèle idéal, l'accélération serait de p et l'efficacité serait de un. En pratique l'efficacité est comprise entre zéro et un, dépendant du degré d'efficacité de l'utilisation des processeurs. L'efficacité est notée E_P et est formulée de la manière suivante :

$$E_P = \frac{A_P}{p}.$$

5.3.4 Coût

Le *coût* de la résolution d'un problème sur un système parallèle est défini comme le produit du temps d'exécution parallèle par le nombre de processeurs utilisés. Ce coût reflète la somme des temps que chaque processeur passe dans la résolution de ce problème. L'efficacité peut dès lors être définie comme le rapport entre le temps d'exécution du meilleur algorithme séquentiel permettant la résolution du problème et le coût sur p processeurs relatif à ce même problème.

Le coût de la résolution d'un problème sur un seul processeur correspond au temps d'exécution du meilleur algorithme séquentiel connu. Un système parallèle est dit de *coût optimal* si le coût de la résolution du problème sur la machine parallèle est proportionnel au temps d'exécution de l'algorithme séquentiel sur un processeur. L'efficacité est donc le rapport entre le coût séquentiel et le coût parallèle et un système parallèle de coût optimal a une efficacité en $\mathcal{O}(1)$.

5.4 Conclusion

Au sein de ce chapitre, nous avons décrit les quatre étapes à suivre lors de l'élaboration d'un algorithme parallèle. Au départ nous avons à notre disposition toutes les spécifications du problème que nous souhaitons paralléliser. Ensuite nous procédons de la manière suivante :

1. Le problème est partitionné en plusieurs parties ou tâches. Ce partitionnement est obtenu en utilisant une décomposition de l'espace ou une décomposition fonctionnelle.

2. Ensuite les communications nécessaires à l'obtention des données utilisées lors de l'exécution des tâches sont établies. Ces communications interviennent quand il y a des dépendances entre les tâches.

3. Puis certaines tâches sont regroupées afin de diminuer les coûts de communication.

4. Ces tâches sont ensuite placées sur les processeurs avec comme objectif la diminution du temps d'exécution.
5. Rajoutons que les entrées/sorties entre la simulation et la visualisation doivent être gérées afin que la visualisation affiche les données calculées par la simulation et que cette dernière puisse interpréter les interactions produites par l'utilisateur lors de l'observation de l'application.

Durant la présentation de cette méthodologie quatre notions fondamentales dans l'élaboration d'algorithmes parallèles ont été introduites :

- La *concurrence*. Elle représente le fait de réussir à exécuter différentes actions simultanément. Ceci est impératif si le programme est destiné à être exécuter sur plusieurs processeur, ainsi que pour obtenir des performances.
- Le *passage à l'échelle*. Il indique la capacité à pouvoir augmenter le nombre de processeurs, ou encore l'espace des données tout en conservant les performances de l'algorithme parallèle.
- La *localité*. Elle représente le rapport entre les accès à la mémoire locale et ceux aux mémoires distantes engendrant des communications. Une gestion correcte de la localité permet l'obtention de hautes performances sur des architectures multiprocesseurs.
- La *modularité*. Elle se réfère à la décomposition d'un problème complexe en sous modules simples, aspect essentiel de toute programmation aussi bien séquentielle que parallèle.

Ce chapitre s'est terminé par la présentation de quelques critères de performance permettant de juger de la qualité de l'algorithme parallèle conçu, mettant en évidence l'utilisation de la part de l'algorithme parallèle des ressources disponibles sur la machine parallèle.

Architectures parallèles 6

Nous souhaitons exécuter notre simulation parallèle de textiles sur une grappe de PCs. Ce chapitre explique la raison de ce choix en analysant les avantages liés à l'utilisation d'une grappe de PCs par rapport aux machines parallèles traditionnelles. Puis nous survolerons les différents modèles et environnements de programmation parallèle permettant de tirer profit de cette puissance de calcul.

6.1 Introduction

Les architectures parallèles offrent une puissance de calcul et une capacité de stockage potentiellement très importantes. Les progrès des composants matériels permettent de disposer de multiprocesseurs très performants quel que soit leur niveau d'intégration : machines parallèles propriétaires (Cray, SGI), grappes autour d'un accélérateur de communication, calcul distribué, etc ...

D'autre part le rapport coût/performance des processeurs et des réseaux à haut débit indique qu'il est économiquement viable aujourd'hui de construire des systèmes à haute performance en inter-connectant des composants standards. Ces systèmes appelés grappes permettent d'obtenir des machines parallèles compétitives par rapport aux systèmes spécialisés avec un coût réduit. Il n'est dès lors plus nécessaire d'avoir recours à des machines multiprocesseurs fort coûteuses pour effectuer des simulations parallèles.

A cet argument financier s'ajoute la possibilité d'obtenir de meilleures performances pour la programmation d'applications irrégulières. En effet, pour bénéficier pleinement des propriétés matérielles offertes par les machines parallèles, les algorithmes doivent être basés sur des opérations régulières de vecteurs denses. Les simulations complexes comportant des structures de données creuses ne sont donc pas efficacement parallélisables

sur de telles machines vectorielles. Il est donc plus avantageux de paralléliser ce type d'applications sur des grappes dont la gestion de l'irrégularité des données est seulement problématique au niveau logiciel et non pas au niveau matériel. Or la partie consacrée à la simulation de textiles a mis en évidence le caractère irrégulier de cette application avec notamment la présence de structure de données creuses. C'est pourquoi il est sans doute préférable d'opter pour l'utilisation d'une machine parallèle de type grappe de PCs standards plutôt que pour une machine vectorielle de type Cray.

De plus, l'implantation de programmes parallèles sur des grappes de machines standards est facilitée par l'apparition de langages de programmation parallèle de haut niveau gérant automatiquement les communications et les synchronisations entre les processeurs ainsi que pour certains l'ordonnancement des tâches de calcul et des données communicables.

Enfin, le dernier point en faveur de l'utilisation d'une grappe plutôt qu'une machine propriétaire concerne la portabilité des programmes. En effet, les applications implantées sur des machines propriétaires ne peuvent généralement pas être exécutées sur une autre machine, prenant en compte les propriétés matérielles de la machine. Or ceci est rarement le cas lors de l'utilisation d'une grappe de machines. En effet, l'un des avantages majeurs étant la possibilité de modifier à volonté la configuration de la grappe par l'ajout ou la suppression de machines. Par conséquence aucune spécificité matérielle n'est généralement prise en compte ou du moins aucune propriété qui nuirait par la suite à la portabilité de l'application.

Dans ce chapitre les différentes architectures parallèles qui existent seront répertoriées en mettant l'accent sur les grappes de PCs, architecture cible de notre application. Puis nous verrons quels sont les modèles de programmation parallèle les plus usuellement employés et nous argumenterons notre choix vis-à-vis de l'un d'entre-eux, à savoir l'environnement de programmation parallèle ATHAPASCAN [116, 104, 102].

6.2 Architectures parallèles

Une machine parallèle consiste en un ensemble de processeurs capables de travailler en coopération afin de résoudre un problème de grosse taille. Elles offrent un potentiel de ressource de calcul important au travers de leurs processeurs, de leurs mémoires, ainsi que de leur bande passante. Cette définition inclut aussi bien les super calculateurs parallèles comportant des centaines ou des milliers de processeurs, que les réseaux de stations de travail ou encore les machines multiprocesseurs.

Les architectures parallèles peuvent être séparées en plusieurs catégories en fonction de la multiplicité des flots d'instructions et de données. Ainsi, Flynn [43] distingue deux types de machines parallèles : *les machines SIMD* (Single Instruction Multiple Data) et *les machines MIMD* (Multiple Instruction Multiple Data). Dans une machine SIMD, tous

les processeurs exécutent la même flot d'instructions sur des données différentes. Cette approche permet de réduire à la fois la complexité du matériel et celle du logiciel mais elle est appropriée seulement pour des problèmes caractérisés par un fort degré de régularité. Dans une machine parallèle MIMD, chaque processeur peut exécuter un flot d'instructions séparé sur ses propres données locales.

Les machines SIMD sont plus simples à utiliser mais elles ne permettent cependant pas de traiter efficacement tous les types de problèmes. Les machines MIMD ont le mérite d'être d'un usage plus général et d'avoir un coût moins élevé. C'est pourquoi la plupart des ordinateurs parallèles sont aujourd'hui des machines MIMD.

Les machines MIMD sont elles-mêmes subdivisées en deux groupes [115], les multiordinateurs et les multiprocesseurs, qui se différencient par la localisation de leur mémoire. Les multiprocesseurs, également appelés *machines MIMD à mémoire partagée*, sont caractérisés par plusieurs processeurs partageant le même espace d'adressage. Tandis que dans le groupe des multiordinateurs ou *machines à mémoire distribuée*, les noeuds qui définissent la machine parallèle sont indépendants les uns des autres, chaque noeud étant composé d'un processeur et d'une mémoire locale. Il existe également des machines à *architecture hybride* qui sont des multiordinateurs de multiprocesseurs.

6.2.1 Machines à mémoire partagée

Les architectures parallèles à mémoire partagée présentent une mémoire contiguë unique accessible directement par leur ensemble de processeurs via un bus ou une hiérarchie de bus (figure 6.1). La plupart des machines à mémoire partagée sont des systèmes symétriques (SMP, Symetric MultiProcessing) où tous les processeurs ont les mêmes fonctions et se disputent les ressources du système de façon uniforme.

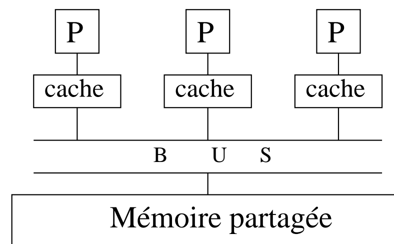


Figure 6.1 *Machine multiprocesseurs à mémoire partagée. P représente un processeur indépendant.*

L'existence d'une mémoire commune entre les processeurs simplifie le travail de parallélisation des algorithmes. La communication entre les processeurs est facilitée par le fait que les communications se font par le biais de lectures et d'écritures de zones de mémoire partagée. Ceci rapproche la logique de construction de programmes de celle de

la programmation séquentielle traditionnelle. D'un autre coté, l'existence d'une mémoire commune est également le point faible de ce type d'architecture en constituant un goulot d'étranglement car tous les processeurs doivent y accéder.

Dans le cas idéalisé du modèle PRAM (Parallel Random Access Machine), souvent utilisé pour étudier d'un point de vue théorique des algorithmes parallèles [103, 12], tout processeur accède en une même quantité de temps tout élément de la mémoire. En pratique, le fait d'augmenter l'architecture introduit généralement une forme de hiérarchie mémoire. En particulier, la contention d'accès à la mémoire partagée peut être réduite en conservant dans le cache de chaque processeur des copies des données fréquemment accédées. En effet, les accès dans les caches des processeurs sont beaucoup plus rapides que ceux effectués dans la mémoire partagée, mais apparaîssent alors des problèmes de cohérence et de synchronisation.

Mais malgré l'emploi de plusieurs niveaux de mémoires caches, ces architectures sont souvent limitées en mémoire d'exécution. Elles ne sont donc pas adaptées au parallélisme à très grande échelle en tant que machine unique.

6.2.2 Machines à mémoire distribuée

Une machine parallèle à mémoire distribuée comporte un ensemble de noeuds reliés par un réseau de communication. Chaque noeud est composé d'un processeur et d'une mémoire locale. La mémoire est ainsi distribuée parmi les processeurs et n'a donc pas de localisation centrale. Toutes les interactions entre les noeuds doivent passer obligatoirement par le réseau. La figure 6.2 illustre ce type de modèle d'architecture parallèle.

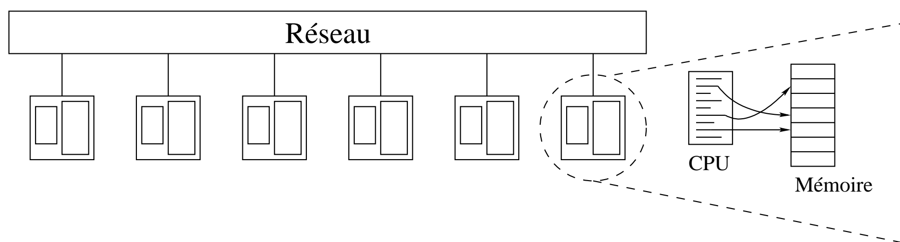


Figure 6.2 *Modélisation d'une machine parallèle MIMD à mémoire distribuée. Chaque noeud est composé d'un processeur et d'une mémoire locale. Les noeuds communiquent entre eux en envoyant et en recevant des messages via le réseau de communication.*

Ce modèle d'architecture présente un *asynchronisme physique* dont il faut tenir compte lors de la programmation, provenant du fait que chaque processeur travaille indépendamment avec sa propre mémoire locale.

De telles architectures peuvent atteindre une taille beaucoup plus importante en terme de nombre de processeurs et de capacité mémoire que celles à mémoire partagée. En

effet il n'y a pas de contention entre les processeurs pour accéder à la mémoire, la seule contrainte d'extension étant la capacité du réseau de communication. Ainsi, des machines composées de plus d'une dizaine de milliers de processeurs ont vu le jour. Ce sont les architectures dites *massivement parallèles* (MPP, Massively Parallel Processing).

6.2.3 Machines graphiques parallèles

En informatique graphique, les machines Silicon Graphics Incorporated (SGI) Origin et Onyx représentent les standards en terme de puissance. Les SGI Origin 2000 [54] sont basées à la fois sur une mémoire partagée et sur un système distribué. Le système est appelé Scalable Shared-Memory Processor (SSMP). Deux processeurs sont connectés via un “hub” à une mémoire partagée pour former un *noeud*. Plusieurs noeuds sont ensuite connectés en utilisant un réseau d'interconnexion afin de former le système Origin. La figure 6.3 illustre cette architecture.

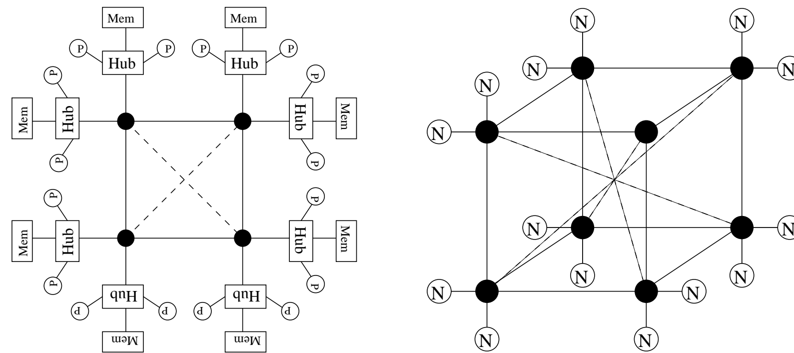


Figure 6.3 *Architecture du système SGI Origin. Deux processeurs sont connectés via un “hub” à une mémoire partagée pour former un “noeud”. Plusieurs noeuds sont ensuite connectés en utilisant un réseau d'interconnexion pour former le système global.*

Au sein de ce système, la mémoire distribuée est vue comme un unique espace mémoire, tout processeur pouvant accéder à tout emplacement de la mémoire. La dernière configuration de cette machine, la SGI Orign 3000, comporte 512 processeurs.

6.2.4 Grappes de machines ou clusters

Depuis quelques années, les *grappes de machines* ou cluster [114] ont fait leur apparition. Une grappe peut se limiter à une grappe d'ordinateurs personnels standards inter-connectés par un réseau haut débit (à gauche de la figure 6.4), mais peut également consister en un système de plusieurs machines *SMP* inter-connectées via un bus de communications et d'entrées/sorties à hautes performances (à droite de la figure 6.4). Ce type

d'architecture est alors dite *hybride*, car elle combine les deux types d'architectures présentées précédemment. Avec cette configuration, nous avons à la fois la mémoire partagée au sein de chaque noeud et la mémoire distribuée entre l'ensemble des noeuds. C'est-à-dire qu'elle profite à la fois de la facilité d'accès aux données offerte par les mémoires partagées, tout en gardant l'extensibilité des machines à mémoire distribuée.

Figure 6.4 *(A gauche) Grappe de PCs du projet iCluster (HP, INRIA, ID-IMAG) composée de 226 mono-processeurs Pentium III à 733 MHz avec 256 Mo de mémoire, inter-connectés via un réseau de 100 Mbit/s . (A droite) Grappe du projet iCluster2 composée de 104 bi-processeurs Itanium-2, 64 bits à 900 MHz avec 3 Go de mémoire, inter-connectés par un réseau Myrinet.*

Cette technologie des grappes s'oppose à celle plus coûteuse des multiprocesseurs en permettant l'obtention à un *moindre coût* d'une machine parallèle ayant un grand nombre de processeurs avec une mémoire d'exécution importante. Ce nombre de processeurs et la taille de cette mémoire étant limités dans les machines multiprocesseurs (512 processeurs pour la SGI Onyx 3000). De plus une grappe a l'avantage d'être *évolutive*. En effet dans une grappe, la puissance globale de calcul peut être augmentée graduellement en ajoutant un autre système standard.

Pour toutes ces raisons, de plus en plus de laboratoires préfèrent investir dans l'acquisition de grappes plutôt que dans l'achat d'une machine multiprocesseurs de type Cray. Les deux grappes présentées sur la figure 6.4 sont testées au sein de notre laboratoire.

6.3 Modèles de programmation parallèle

Après avoir effectué un bref survol des modèles de machines parallèles, nous allons dans cette partie étudier les modèles de programmation parallèle les plus usuellement employés. En effet une machine parallèle haute performance n'est intéressante que si nous arrivons à tirer profit de toutes ses ressources matérielles. Les modèles de programmation parallèle permettent d'établir des liens entre le développement d'une application sur un type d'architecture précis tout en faisant abstraction des détails techniques qui peuvent varier.

Ces modèles peuvent être classés selon différents niveaux d'explicitation du parallélisme [113, 42] :

- *Modèle à parallélisme implicite.* Toutes les activités liées à l'exécution parallèle sont complètement cachées au programmeur. La parallélisation est obtenue par transformation du programme séquentiel en programme parallèle par des compilateurs parallélisateurs (PROLOG Parallèle [41]).
- *Modèle à parallélisme explicite.* Le parallélisme est décrit de façon explicite sans pour autant définir la façon dont l'application est divisée en tâches, ni l'ordonnancement de celles-ci et ni leurs communications (Multilisp [61]).
- *Modèle à décomposition explicite.* Les langages offrent des primitives explicites pour définir les tâches à exécuter en parallèle. La réalisation de l'ordonnancement et la gestion des communications sont par contre transparentes pour le programmeur (ATHAPASCAN [116, 104], Cilk [67, 24]).
- *Modèle à placement explicite.* Les programmeurs prennent en charge la décomposition en différentes tâches et réalisent leur placement sur les processeurs. Par contre, les communications et les synchronisations sont transparentes (Linda [4]).
- *Modèle guidé par événements.* Les mouvements de données et les synchronisations ne peuvent être effectuées que par la production d'un événement préétabli lequel déclenche les traitements (ACTORS [2, 3]).
- *Modèle à parallélisme totalement explicite.* Tous les aspects de décomposition, d'ordonnancement, de communication et de synchronisation sont à la charge du programmeur de l'application parallèle.

Une parallélisation implicite effectuée à partir d'un programme séquentiel demande beaucoup moins d'efforts que l'établissement d'une parallélisation totalement explicite. Par contre son utilisation en dehors du cadre des architectures à mémoire partagée est souvent difficile. Dans le cadre des architectures distribuées, une parallélisation totalement explicite est la seule envisageable pour obtenir de bonnes performances. En effet la résolution d'un problème complexe nécessite une vue globale de celui-ci, or les outils automatiques utilisés par les autres modèles ne peuvent avoir qu'une vision locale liée à un algorithme séquentiel particulier. La parallélisation de notre simulation de textiles est basée sur un modèle explicite.

La source du parallélisme constitue un autre critère permettant de classer le modèles de parallélisme. Deux groupes de modèles peuvent alors être distingués : *le parallélisme de données* et *le parallélisme de contrôle*. Le premier groupe est approprié à la programmation de machines SIMD tandis que le second s'adapte aux machines MIMD.

6.3.1 Parallélisme de données

Le parallélisme de données est appelé ainsi car il exploite la concurrence qui résulte de l'exécution de la même opération sur des éléments différents d'une structure de données. Dans ce mode d'expression du parallélisme, le travail des processeurs est guidé par la distribution des données. Les communications sont définies comme des phases de redistribution des données sur les processeurs. Le programme parallèle est une succession de phases de calculs et de phases de communications pour la redistribution des données. Cette source de parallélisme est souvent utilisée lors d'opérations sur des vecteurs, fréquentes dans les problèmes d'algèbre linéaire dense. Elle apparaît dans notre simulation quand les mêmes calculs sont effectués sur des données différentes comme par exemple le calcul des accélérations de chacune des particules.

6.3.2 Parallélisme de contrôle

Le parallélisme de contrôle consiste à décrire un algorithme parallèle sous la forme d'un graphe orienté sans cycle. Les noeuds du graphe sont des suites d'opérations élémentaires exécutées séquentiellement, ce sont les tâches du graphe. Les arcs du graphe indiquent des contraintes de précédence entre les taches. C'est-à-dire l'utilisation par une tâche d'une donnée calculée par une ou plusieurs tâche(s) précédente(s). Ce graphe appelé *graphe de précédence* ou *graphe de tâches*, définit un ordre partiel sur les tâches. Les tâches non ordonnées par cet ordre partiel peuvent être exécutées en parallèle. Nous verrons dans la section 7.4.3, comment l'environnement de programmation parallèle ATHAPASCAN construit ce graphe de précédence relatif à notre application en cours d'exécution.

Le parallélisme de données peut être considéré comme un cas particulier de parallélisme de contrôle. Dans le parallélisme de contrôle, plusieurs paradigmes de programmation sont usuellement employées. Nous allons maintenant étudier trois d'entre eux : le modèle à mémoire partagée, le modèle par échange de messages et le modèle à mémoire virtuelle partagée.

Modèle à mémoire partagée

La programmation par mémoire partagée est adaptée à l'utilisation de machines SMP, c'est-à-dire à l'emploi de machines multiprocesseurs. Les communications sont faites directement par *variables partagées* contrôlées par des mécanismes de synchronisation

(sémaphores, et verrous par exemple). La plupart des systèmes d'exploitation existants pour ce type de machine mettent à disposition une interface de programmation implantant ce modèle. Son inconvénient est le fait d'être trop lié à un style d'architecture de machine, réduisant ainsi la portabilité des programmes.

Modèle par échange de messages

L'échange de messages est le modèle de base qui permet la réalisation de communications entre processeurs d'une architecture à mémoire distribuée. Son implantation est effectuée à l'aide de primitives d'*envoi* et de *réception*, dont les paramètres sont l'identificateur du processeur partenaire pour la communication ainsi que le message à envoyer.

Dans ce modèle, le problème du contrôle du flux des messages est très important. Deux sémantiques existent : *synchrone* (avec rendez-vous) et *asynchrone* (canal de communication borné). Pour une communication synchrone, émetteur et récepteur doivent demander à communiquer pour que la communication ait lieu. Dans les communications asynchrones, l'émetteur fait son émission à n'importe quel moment et le message sera mémorisé et délivré au récepteur quand celui-ci le demandera. La plupart des bibliothèques de communication offrent un modèle à canal borné et laissent à la charge du programmeur le calcul de la taille des tampons stockant les messages, nécessaires à son application pour éviter les blocages.

Les environnements de programmation qui implantent ce modèle de communication (MPI, PVM) permettent d'homogénéiser les caractéristiques techniques de tous les noeuds présents au sein d'une grappe de machines. La couche de communication de l'environnement de programmation parallèle ATHAPASCAN, appelée INUKTITUT [83], est basée sur ce même concept d'homogénéisation.

6.4 Environnements de programmation parallèle

Après avoir décrit quelles étaient les différentes architectures parallèles qui existaient et quels modèles de programmation parallèle permettaient de faire les liens entre le développement d'une application et ces types de machines, nous présentons dans cette section les environnements de programmation parallèle reposant sur ces modèles qui ont été conçus afin de faciliter l'exploitation des sources de parallélisme présentes dans une application. Certains sont basés sur le paradigme de passage de messages (PVM et MPI), un autre sur le modèle à mémoire partagée (Cilk) et le dernier sur le concept de la mémoire virtuelle partagée (ATHAPASCAN).

6.4.1 Échange de messages : PVM et MPI

Le modèle par échange de messages a donné de nombreuses variantes. Deux environnements de programmation parallèle populaires s'inspirent de ce modèle. Ils s'agit

de la bibliothèque PVM (Parallel Virtual Machine) [51] et de l'interface de passage de messages MPI (Message Passing Interface) [129].

Bibliothèque de passage de messages PVM

La bibliothèque de passage de messages PVM est une interface de programmation simple et facile d'utilisation. C'est pourquoi de nombreuses applications scientifiques utilisent à ce jour ce système. Son développement a débuté en 1989 au sein du Laboratoire National Oak Ridge. Le premier prototype a été développé par Vaidy Sunderam et Al Geist. Son développement a ensuite continué à l'université du Tenessee et la diffusion de la version 2 de PVM a commencé en mars 1991.

La bibliothèque PVM permet d'appréhender un ensemble de systèmes hétérogènes comme une unique machine parallèle virtuelle. PVM gère de façon transparente le routage de messages, la conversion de données et l'ordonnancement de tâches au sein d'un réseau élaboré à partir de machines hétérogènes. L'utilisateur décrit son application comme un ensemble de *tâches* coopérantes. La bibliothèque PVM fournit un ensemble de routines permettant aux tâches d'accéder aux ressources PVM. Ces routines permettent l'initialisation et la terminaison des tâches ainsi que les communications et les synchronisations entre ces tâches. Il est également possible de communiquer des structures de données particulières.

A tout moment de l'exécution, une tâche peut démarrer ou stopper une autre tâche, ou encore ajouter ou supprimer un ordinateur de la machine virtuelle. Tout processus peut communiquer et/ou se synchroniser avec un autre. De manière générale, le système PVM est basé sur les principes suivants :

– Les tâches de calcul d'une application s'exécutent sur un ensemble de machines sélectionnées par l'utilisateur lors de l'exécution du programme PVM. Ce pool de machines peut être constitué aussi bien de machines mono processeur que multiprocesseurs (incluant des machines à mémoire partagée et à mémoire distribuée). Il est possible de retirer ou d'ajouter des machines au sein de ce pool durant l'exécution permettant une tolérance aux pannes.
– Les applications peuvent soit percevoir l'environnement matériel comme un ensemble de processus virtuels ou peuvent choisir d'exploiter les capacités des machines spécifiques en positionnant certaines tâches de calcul sur la machine la plus appropriée.
– L'unité de parallélisme dans PVM est une tâche, c'est-à-dire un processus séquentiel de contrôle qui alterne entre communications et calculs. Plusieurs tâches peuvent être exécutées sur un unique processeur.
– Des ensembles de tâches de calculs, chacun effectuant une partie des calculs de l'application, coopèrent en envoyant et recevant des messages entre eux. La taille des messages est seulement limitée par la quantité de mémoire disponible.
– PVM supporte une hétérogénéité en terme de machines, réseaux et applications.

Les messages transitant entre les machines ayant des représentations de données différentes, peuvent contenir des données de types différents.
- PVM supporte la multi programmation permettant de tirer profit des ressources matérielles fournies par les machines multiprocesseurs.

La machine PVM est en fait constituée de deux parties. La première est un *démon*, appelé *pvmd3* ou *pvmd*, résidant sur toutes les machines permettant d'émuler la machine virtuelle. La seconde partie de PVM est la bibliothèque de routines. Celle-ci contient toutes les fonctions de base permettant la coopération entre les tâches d'une application. La bibliothèque PVM permet donc l'utilisation de plate-forme hétérogène avec une tolérance aux pannes grâce à la possibilité de supprimer ou d'ajouter des machines en cours d'exécution.

Mais ces deux avantages n'ont pas permis l'obtention de performances suffisamment intéressantes. C'est pourquoi d'autres chercheurs ont commencé à développer une autre bibliothèque de passages de messages, MPI, plus légère et ainsi plus facile à optimiser sur une architecture donnée.

Interface de passage de messages MPI

La version originale du standard MPI [129] a été créée par le Message Passing Interface Forum (MPIF) et la première version publique a été diffusée en 1994. Dans l'approche adoptée pour la programmation parallèle par la bibliothèque de passage de messages MPI, un ensemble de processus exécutent un programme écrit dans un langage séquentiel dans lequel ont été rajoutés des appels aux différentes fonctions contenues dans la bibliothèque permettant l'envoi et la réception de messages.

Au sein d'un calcul écrit en MPI, plusieurs processus communiquent ensemble via ces appels aux routines de la bibliothèque. Dans la plupart des implantations de MPI, un nombre fixé de processus sont créés à l'initialisation, et un processus est généralement créé par processeur. Le modèle de programmation parallèle MPI est souvent référencé en tant que SPMD (Single Program Multiple Data, programme unique données multiples) dans lequel tous les processus exécutent le même programme, à distinguer des modèles MPMD (Multiple Program Multiple Data, programmes multiples données multiples) dans lesquels les processus peuvent exécuter des programmes distincts.

Les processus peuvent utiliser des opérations de *communication point-à-point* pour envoyer un message depuis un processus nommé vers un autre. Un groupe de processus peut également employer des opérations de *communications collectives* bien utiles lors d'opération de diffusion ou de somme. La bibliothèque MPI permet également l'utilisation de *communications asynchrones*. De plus MPI supporte la programmation modulaire. Un mécanisme appelé *communicator* permet au programmeur MPI de définir des modules qui encapsulent des structures de communications internes.

L'approche adoptée par MPI pour la programmation parallèle est donc entièrement basée sur un échange de communications entre les processus. De nombreuses routines sont mises à la disposition du développeur pour effectuer différents types d'envois et de réceptions de messages (synchrones, asynchrones, collectifs, point-à-point) ainsi que des opérations globales (barrière, diffusion, réduction). Par contre MPI ne spécifie pas :

- les opérations relatives à l'utilisation d'une mémoire partagée ;
- les opérations nécessitant des appels systèmes non prévus lors de la spécification du standard MPI (exécutions à distance, messages actifs, interruption de réception) ;
- des outils d'aide à la programmation ;
- des outils de déboguages ;
- un support pour la multiprogrammation à base de processus légers ;
- la gestion des tâches ;
- des fonctions d'entrées/sorties.

Certaines de ces spécifications devraient être adoptées dans la nouvelle version MPI-2, toujours en cours de développement, qui représente en fait la combinaison des interfaces MPI et PVM.

Bilan des environnements par processus communicants

Les environnements de programmation parallèle basés sur le modèle de passage de messages se révèlent fastidieux à utiliser quand il s'agit de paralléliser des applications complexes et irrégulières. En effet le programmeur doit gérer à la fois les communications effectuées entre processeurs, ainsi que les synchronisations ou encore l'ordonnancement des tâches de calculs et le placement des données parmi les processeurs. Or nous verrons dans le chapitre 8 que les communications engendrées par la simulation de textiles sont complexes et qu'il serait impensable de les gérer à la main.

6.4.2 Mémoire partagée : Cilk

Cilk [67, 24] est un langage destiné à la programmation de machines parallèles à mémoire partagée dont le développement a débuté en 1993 au MIT. C'est une extension du langage C, offrant des primitives pour l'expression du parallélisme de contrôle par création explicite de tâches. Cilk est ainsi un langage de programmation parallèle multiflots, spécialement conçu pour exploiter du parallélisme dynamique et hautement asynchrone, difficile à écrire via le parallélisme de données ou le passage par messages.

La philosophie de Cilk est de laisser le développeur se concentrer sur la structure du programme afin de déterminer les sources du parallélisme de son application et d'exploiter la localité. Pour cela le système Cilk gère l'ordonnancement des calculs afin d'obtenir une exécution efficace sur une architecture cible. D'autre part l'emploi de cet environnement parallèle est facilité par le fait que le langage de programmation multiflots de Cilk

est basé uniquement sur trois mots-clés [55]. De plus un modèle de coût, permettant de garantir les efficacités en temps et en consommation mémoire, est associé au modèle de programmation.

C'est cependant à l'utilisateur de garantir explicitement à l'aide de synchronisations simples qu'aucune contrainte de précédence n'apparaît dans son application. En effet l'utilisateur est libre d'accéder aux données partagées à tout moment, c'est-à-dire que Cilk ne garantit pas le fait qu'une donnée partagée en cours d'écriture ne peut être lue ou modifiée en même temps par un autre processus.

6.4.3 Mémoire virtuelle partagée : ATHAPASCAN

L'environnement de programmation parallèle ATHAPASCAN [116, 104] est développé au sein du laboratoire ID-IMAG (Informatique et Distribution) dans le cadre du projet INRIA-APACHE. Son développement a débuté en 1993. Cet environnement est comparable à Cilk du point de vue de l'utilisateur. C'est une interface de programmation parallèle de *haut niveau* au sens où aucune référence n'est faite par rapport au support d'exécution. De plus, ATHAPASCAN est un *langage explicite* au sens où le parallélisme est exprimé par des tâches créées à l'aide de deux mots-clés. Enfin, il gère l'ordre d'exécution des tâches et les communications entre processeurs.

Mais par rapport à Cilk, il a l'avantage d'assurer une portabilité bien supérieure grâce à la possibilité d'utiliser des plates-formes parallèles distribuées. En fait ATHAPASCAN a la propriété de combiner deux concepts, le passage de messages et la mémoire partagée, permettant d'exploiter les deux types d'architectures en même temps, à mémoire partagée et à mémoire distribuée. De plus ATHAPASCAN garantit la sémantique d'accès aux données de la mémoire partagée grâce à des synchronisations implicites entre les tâches. En effet chaque tâche déclare les accès effectués sur une mémoire partagée fournie par la bibliothèque.

Cet environnement est en fait composé de deux modules complémentaires : INUKTITUT et ATHAPASCAN. INUKTITUT, l'extension d'Athapascan-0 [26, 83], est le module exécutif qui permet l'utilisation de la multiprogrammation légère dans un contexte distribué. Ce module, basé sur un couplage entre différentes bibliothèques de processus légers (threads POSIX) et de communication standards (Message Passing Interface), constitue le noyau exécutif de l'environnement et assure sa portabilité. INUKTITUT permet de gérer les communications effectuées entre processus, sans se soucier du type de réseau utilisé. ATHAPASCAN, anciennement appelé Athapascan-1 [116, 104, 102], est l'interface applicative (API) au-dessus d'INUKTITUT dont nous nous servons. Il implémente un modèle de programmation de haut niveau basé sur un mécanisme de création de tâches collaborant entre elles par l'accès à une mémoire virtuelle partagée.

Pour pouvoir employer cet environnement il suffit de découper l'application séquentielle en plusieurs *tâches de calculs* plus ou moins indépendantes et de déterminer quels seront les *objets partagés* par les différentes machines constituant la grappe de calcul.

Les tâches sont alors exécutées dès qu'elles sont prêtes, c'est-à-dire dès que toutes les données dont elles se servent sont disponibles. ATHAPASCAN crée le *graphe de flots de données* associé au programme afin d'exécuter ces tâches dans le respect des contraintes de précédence qui existent entre elles. D'autre part ces tâches sont exécutées de façon à optimiser leur exécution en employant des ordonnancements adaptées à l'application.

Cet environnement met également à la disposition de ses utilisateurs des outils de déboguages comme par exemple l'affichage du graphe de dépendances associé au programme, ainsi que la visualisation du placement des tâches sur les processeurs. Un logiciel permettant de visualiser des traces systèmes est également associé à cet environnement. Il s'agit du logiciel Pajé [34, 33, 35] développé également au sein du laboratoire ID-IMAG. Il permet notamment l'observation postmortem des communications qu'il y a eu lieu entre les processus, ainsi que leurs états tout au long de la simulation (en attente, en cours d'exécution).

Pour toutes ces raisons, nous pensons qu'ATHAPASCAN est l'environnement de programmation parallèle le mieux adapté à l'exploitation des ressources de calculs fournies par une grappe de machines mono-processeurs ou multiprocesseurs. Les détails de ses spécifications sont fournis dans le chapitre suivant.

6.5 Conclusion

Nous avons vu qu'il existait plusieurs types d'architectures parallèles qui se différencient par la localisation de la mémoire. Pour notre part, nous avons choisi d'utiliser une grappe de multiprocesseurs, combinant une architecture à mémoire partagée et une architecture à mémoire distribuée. Ce type d'architecture a l'avantage d'être évolutif à un moindre coût.

Ensuite, nous avons présenté les modèles de programmation parallèle permettant d'exploiter ces architectures. Le modèle d'échange de messages est employé sur les machines distribuées, tandis que le modèle à mémoire partagée sert sur les machines ayant ce type de mémoire. Des environnements de programmation parallèle basés sur ces modèles aident les programmeurs à implanter leur application sur ces machines parallèles. Deux choix sont alors possibles : utiliser un langage de passage par messages ou bien utiliser un langage de programmation de haut niveau.

Les bibliothèques de communication MPI et PVM utilisent le paradigme de passage de messages et sont très largement employés dans la communauté scientifique. Ces deux langages sont très proches l'un de l'autre [52]. PVM sera préféré lors de l'utilisation de plates-formes hétérogènes. De plus, grâce à la possibilité d'arrêt ou de démarrage d'un processus en cours d'exécution, il offre une tolérance aux pannes importante. MPI quant à lui sera préféré pour ses performances. En effet la majorité des constructeurs de machines parallèles ont fait le choix de MPI, plus léger que PVM (oubliant le coté hétérogénéité), et

plus facile à optimiser pour une architecture cible. De plus, MPI apporte un support non négligeable pour les communications de type collectives. La nouvelle version de MPI, MPI-2, est toujours en cours de développement. Cette version devrait être une fusion de PVM et MPI.

Mais ces bibliothèques s'avèrent limitées quand il s'agit de paralléliser des applications complexes dont la gestion des communications est ardue. Il convient alors d'employer des langages de plus haut niveau, gérant en autres les communications entre processus. Le langage Cilk offre un tel support à la programmation parallèle en exprimant le parallélisme d'une application via la création de tâches effectuée à l'aide de quelques mots-clés. Mais ce langage est seulement dédié aux machines multiprocesseurs. Il n'est donc pas utilisable sur une machine parallèle de type grappe.

Nous avons alors présenté ATHAPASCAN, un autre environnement de haut niveau, qui sait tirer profit à la fois de la mémoire partagée et de la distribution des processeurs présentes sur une grappe. D'une part, ATHAPASCAN, à la différence de Cilk, gère également les synchronisations entre les processus. D'autre part, cet environnement sait tirer partie de l'évolutivité des grappes en rendant les applications très portables (exécutables sur tout type d'architecture : séquentielle, SMP, grappe de PCs, grappe de SMPs) et en leur permettant ainsi de passer à l'échelle. C'est pourquoi notre choix s'est porté sur cet environnement que nous présentons plus longuement dans le chapitre suivant.

ATHAPASCAN 7

La parallélisation de la simulation de textiles est effectuée en utilisant l'environnement de programmation parallèle ATHAPASCAN. *Ce chapitre présente d'une part l'interface de programmation de cet environnement : la définition, la création et l'exécution des tâches de calcul ainsi que les droits d'accès possibles aux données partagées prises en paramètres de ces tâches ; et d'autre part le modèle d'exécution de cet environnement : l'établissement de l'ordre d'exécution des tâches ainsi que le support d'exécution.*

7.1 Introduction

Notre méthode de parallélisation est basée sur un partitionnement du problème initial en sous-tâches de calculs comme l'illustrait la figure 5.1 de la section 5.2.1. Certaines de ces tâches de calculs présentent des dépendances de données en raison des interactions qui existent entre particules voisines. L'exécution parallèle étant réalisée sur une machine parallèle de type grappe, ces tâches vont être distribuées sur des machines différentes. Les interactions vont ainsi engendrer des communications pour effectuer les transferts de données d'une machine à l'autre.

Il peut être facile d'élaborer ces communications avec un modèle de type échange de messages (MPI, PVM) pour un problème de petite taille sur un nombre réduit de machines, mais dès que le nombre de tâches devient important ainsi que le nombre de machines, la gestion de ces communications devient vite difficile. La figure 7.1 présente le graphe de flots de données associé à la simulation de textiles avec un découpage du problème en seulement 16 tâches. Le schéma des communications se révèle déjà complexe pour un nombre de tâches relativement faible.

C'est pourquoi, pour pouvoir utiliser plus facilement les ressources disponibles sur

la grappe de machines, nous avons choisi d'utiliser l'environnement de programmation parallèle ATHAPASCAN [116, 104], développé dans le laboratoire ID-IMAG dans le cadre du projet INRIA-APACHE. ATHAPASCAN offre un modèle de programmation parallèle haut niveau basé sur une mémoire virtuelle partagée. Les avantages principaux de cet environnement sont sa facilité de programmation, la portabilité des programmes et la gestion automatique des communications et des synchronisations, ainsi que l'aide à la mise en oeuvre d'un bon équilibrage de la charge sur l'ensemble des processeurs. Notre application codée en ATHAPASCAN pourra notamment être exécutée sur n'importe quel type de machines parallèles sans aucune modification du programme.

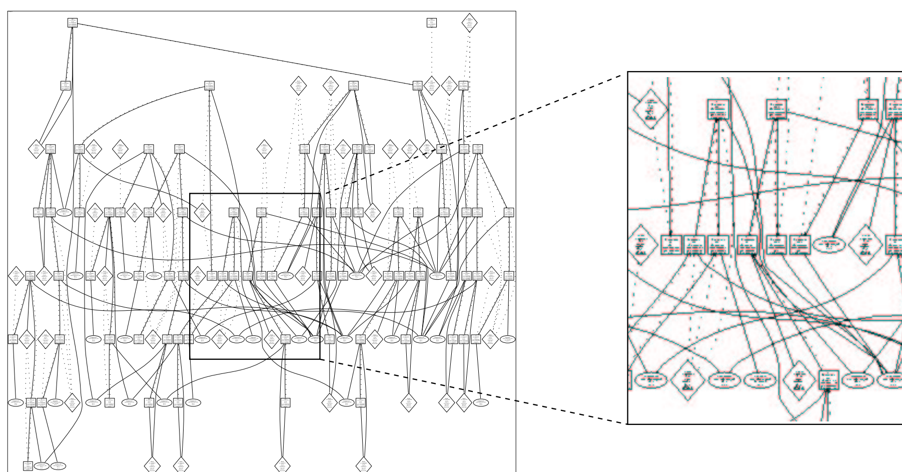


Figure 7.1 *Graphe de flots de données de la simulation de textiles. Les rectangles, les losanges et les cercles représentent respectivement les accès aux données partagées, les données partagées et les tâches de calculs. Les traits représentent les dépendances.*

Reprenons la figure 5.1 de la section 5.2.1, page 62. Les quatre premières étapes de l'élaboration de l'algorithme parallèle sont : (1) l'élaboration du partitionnement du problème initial en tâches de calcul de taille minimale, (2) l'établissement des communications entre les tâches générées par leurs interactions, (3) l' agglomération des tâches entre elles afin d'augmenter la granularité de la parallélisation, (4) le placement des données sur les processeurs ainsi que l'ordonnancement des tâches.

L'environnement parallèle ATHAPASCAN se charge d'effectuer les étapes 2 et 4 de cette méthodologie. En effet après avoir partitionné le problème en plusieurs tâches de calculs en spécifiant les données dont elles auront besoin, ATHAPASCAN est capable dans un premier temps de générer le graphe de dépendances de notre application. Ensuite à partir de ce graphe, ATHAPASCAN place les données sur les processeurs en suivant la politique que nous lui avons spécifiée. Enfin il ordonnance les tâches en minimisant les communications également totalement prises en charge par l'environnement.

L'utilisation de l'environnement ATHAPASCAN permet ainsi de se concentrer davantage sur la parallélisation des applications que sur leur mise en oeuvre.

Ce chapitre présente d'une part l'interface de programmation de l'environnement ATHAPASCAN et d'autre part le modèle d'exécution de cet environnement de programmation parallèle. Nous allons donc étudier comment les tâches de calculs issues du découpage du problème initial sont créées dans ATHAPASCAN. Puis nous verrons comment leurs dépendances sont spécifiées. Enfin nous détaillerons les mécanismes internes d'ATHAPASCAN.

7.2 Programme ATHAPASCAN

Un programme ATHAPASCAN n'est autre qu'un programme écrit dans le langage C++ à l'intérieur duquel deux mots-clés ont été rajoutés sous la forme de `template` : `Fork` (pour la création des tâches) et `Shared` (pour la spécification des données partagées). Nous expliciterons par la suite plus longuement ces deux concepts. Par conséquent, le langage ATHAPASCAN est compilable par n'importe quel compilateur C++ standard facilitant ainsi son utilisation.

7.3 Tâches et objets partagés

Un programme ATHAPASCAN peut être vu comme un ensemble de *tâches* créées dynamiquement pouvant *partager* des objets. Dans le cadre de la simulation de textiles, les tâches correspondent aux calcul des forces, des accélérations, des schémas d'intégration explicites ou implicite, tandis que les données partagées correspondent à des sous-ensembles comportant les caractéristiques des particules. Ces tâches sont ensuite exécutées de manière *asynchrone* par ATHAPASCAN.

La granularité d'un algorithme implanté avec l'environnement ATHAPASCAN est définie par la taille des données mises en paramètres de ces tâches. Elle est ainsi explicitement fournie par le programmeur lors de la *création des tâches*. Dans notre cas, les paramètres des tâches étant les blocs de particules issus de la décomposition, la granularité de notre algorithme parallèle correspondra ainsi à la taille de ces blocs.

7.3.1 Tâche ATHAPASCAN

Une tâche au sens ATHAPASCAN consiste en l'association d'une procédure et de paramètres. Ces tâches sont dynamiquement créées lors de l'exécution. Une tâche ATHAPASCAN (création + exécution) peut donc être vue comme un appel de procédure standard. La seule différence réside dans l'asynchronisme de l'exécution de la tâche créée, c'est-à-dire que le créateur de la tâche n'attend pas la fin de l'exécution de cette tâche pour continuer

sa propre exécution. Deux types de passages des paramètres sont possibles au sein de ces tâches :

– par *valeur* : la tâche possède une copie privée des paramètres. Aucune modification locale accomplie sur ces paramètres ne pourra être vue par une autre tâche.
– par *référence* : la tâche possède une référence sur les objets partagés relatifs aux paramètres. Par conséquent, plusieurs tâches peuvent accéder au même objet partagé. Dans ce contexte, une version de l'objet partagé est fournie par le système à chaque tâche l'accédant. Une version correspond en réalité à un identifiant permettant au système d'ordonnancer l'exécution des tâches dans un ordre assurant la cohérence des données.

Un programme ATHAPASCAN peut donc être vu comme un ensemble de tâches ordonnancées par le système et distribuées sur les noeuds lors de son exécution. Nous verrons ultérieurement comment ATHAPASCAN élabore l'ordonnancement et le placement de ces tâches sur l'ensemble des processeurs de la machine parallèle cible.

7.3.2 Procédure associée à une tâche

La définition d'une tâche ATHAPASCAN correspond à celle d'un objet C++. Cet objet est défini par une classe fournissant uniquement la méthode `void operator()`. L'algorithme 10 illustre cette définition.

Algorithme 10 Définition de la procédure associée à une tâche ATHAPASCAN

```
1 :   struct user_task {
2 :         void operator() ( [parametres] ) {
3 :            ...
4 :         }
5 :   };
```

Dans cet algorithme, la tâche `user_task` est définie par l'ensemble des instructions contenues dans la méthode `operator()`. La granularité du programme correspondra à la taille des paramètres de cette méthode.

7.3.3 Création et exécution d'une tâche

L'algorithme 11 indique comment la tâche `user_task` définie précédemment peut être exécutée de manière séquentielle (ligne 2) ou bien en parallèle via ATHAPASCAN (ligne 5). Le mot-clé `Fork` est utilisé par ATHAPASCAN pour définir les tâches qui seront parallélisées.

Algorithme 11 Exécution de la procédure associée à une tâche ATHAPASCAN

```
1 :   // Appel sequentiel de la fonction C++ user_task
2 :   user_task() ( [parametres] ) ;
3 :   // user_task est alors executee selon l ordre sequentiel

4 :   // Creation de la tache ATHAPASCAN user_task
5 :   a1::Fork< user_task > ( [sched_attributes] ) ( [parametres] ) ;
6 :   // user_task est alors executee de maniere asynchrone
7 :   // Les synchronisations sont definies par les acces aux donnees partagees
8 :   // La semantique respecte l ordre de reference
```

En résumé, pour *"Forker"* une tâche il faut :

- écrire le code qui sera parallélisé en surchargeant l'opérateur `operator()` de la classe à "Forker" (algorithme 10),
- appeler la tâche en utilisant le mot-clé `Fork` (algorithme 11),
- utiliser des paramètres qui sont communicables au sens d'ATHAPASCAN, c'est-à-dire des objets pour lesquels les opérateurs « et » auront été surchargés. Ces paramètres peuvent être soit des objets ou variables réguliers, soit des données partagées (`Shared`) utilisées par différentes tâches (section 7.3.4).

En bref, toutes les instructions écrites à l'intérieur de la procédure de la tâche sont exécutées en parallèle par ATHAPASCAN à la condition que le mot-clé `Fork` ait été mis devant l'appel de cette tâche. L'utilisateur d'ATHAPASCAN définit ainsi toutes les tâches de son application, en mettant comme paramètres des procédures toutes les données qui sont nécessaires aux tâches pour effectuer leurs calculs. D'autre part il est important de noter que l'appel de ces fonctions est programmé dans le même ordre que celui qui aurait été employé en séquentiel, seul le mot-clé `Fork` a été rajouté.

Par ailleurs, pour que ATHAPASCAN sache si des tâches peuvent être exécutées en concurrence ou au contraire ne peuvent être exécutées que les unes après les autres, le développeur de l'application doit faire attention aux types d'accès qu'il autorise sur les données. En effet c'est la spécification de ces accès qui va permettre à ATHAPASCAN de définir les dépendances qui existent entre les tâches.

7.3.4 Droits d'accès des données partagées

Une tâche ATHAPASCAN n'a accès qu'aux données qui lui sont passées en paramètres. Ces données peuvent être partagées par différentes tâches. Elles sont alors déclarées à l'aide du mot-clé `Shared`. Plusieurs types d'accès existent selon si la tâche doit seulement lire la donnée partagée ou au contraire doit pouvoir la modifier. ATHAPASCAN propose quatre types d'accès différents à un objet communicable de type T :

en lecture seule (read)	`a1::Shared_r< class T >`
en écriture seule (write)	`a1::Shared_w< class T >`
en lecture/écriture (read/write)	`a1::Shared_r_w< class T >`
en écriture cumulée (cumulative write) selon une fonction de cumulation F	`a1::Shared_cw< class F, class T >`

TAB. 7.1 *Les différents droits d'accès possibles pour les variables partagées* `a1::Shared`

Ces droits d'accès sur les données partagées vont définir les dépendances entre les tâches possédant des paramètres communs :

– La *lecture* et l'*écriture* d'une donnée partagée peut être effectuée en concurrence par plusieurs tâches en même temps. Dans le cas de l'écriture, la valeur finale de la donnée est la dernière qui lui a été attribuée par rapport à l'ordre de référence.
– Dans le cas d'une donnée partagée accédée en *lecture/écriture*, la tâche peut mettre à jour directement cette donnée qui peut être lue ou modifiée au sein même de la tâche. Deux tâches accédant à la même donnée en lecture/écriture ne pourront s'exécuter en même temps. L'ordre lexicographique est appliqué pour savoir quelle est la tâche qui dépendra de l'autre et en conséquence devra atteindre la fin de l'exécution de la première.
– Et enfin, dans le cas d'un accès en *écriture cumulée*, la donnée partagée ne peut être que cumulée selon la définition d'une fonction cumulative F qui lui est associée. La valeur résultat de cette donnée cumulée correspond à l'accumulation de toutes les autres valeurs écrites par un appel à cette fonction, c'est-à-dire à l'accumulation des résultats obtenus par toutes les tâches ayant cette donnée en écriture cumulée. Cette accumulation peut donc être effectuée en concurrence par plusieurs tâches à la seule condition que F soit une fonction *commutative des paramètres* et *associative*.

7.3.5 Accès mémoire possibles pour une tâche

Chaque tâche ATHAPASCAN peut accéder à deux niveaux de mémoire :

– A la *pile*, la mémoire locale de la tâche. Cette mémoire locale contient les paramètres et les variables locales (c'est une pile classique au sens C++). Cette pile est automatiquement libérée lors de la fin de la tâche.
– Au *tas*, la mémoire locale du noeud (processus UNIX) qui exécute la tâche. Les objets sont alloués et libérés dans ou depuis le tas directement à partir des primitives C/C++ (malloc, free, new, delete). Toutes les tâches exécutées sur un même noeud partagent ce tas mémoire. En conséquence si une tâche ne libère pas les objets alloués dans le tas, la tâche suivante risque d'avoir moins de mémoire que prévu sur le noeud.

Pour récapituler, afin de paralléliser le programme d'une application en utilisant l'environnement ATHAPASCAN, l'utilisateur décompose son application en tâches qui potentiellement pourront être exécutées sur des processeurs différents. Par conséquent, si nous souhaitons que plusieurs tâches connaissent une même donnée, celle-ci devra être définie comme une variable partagée. De même si nous souhaitons récupérer le résultat obtenu par une tâche, celui-ci doit être placé dans une variable partagée, sinon il ne sera connu qu'au sein de la tâche. De plus les variables partagées ne peuvent être modifiées ou simplement consultées qu'à l'intérieur d'une tâche qui définira les droits d'accès sur cette donnée.

7.4 Modèle d'exécution

Nous allons désormais détailler les mécanismes internes de cet environnement. Comment ATHAPASCAN va gérer les dépendances entre les tâches ? Comment il va placer les données sur les processeurs ? Comment les tâches vont être ordonnancées ? En bref, comment l'exécution parallèle va se dérouler sur la machine parallèle ?

7.4.1 Garanties fournies par ATHAPASCAN

L'exécution d'un programme ATHAPASCAN se base sur l'ordre de référence : toute lecture voit la dernière écriture dans l'ordre de référence de l'exécution (proche de l'ordre séquentiel) quelque soit le site d'exécution des tâches. De plus, le système se charge de migrer les données nécessaires à l'exécution des tâches.

Par ailleurs, le système assure à l'utilisateur que l'exécution d'une tâche est effectuée en respectant les contraintes de synchronisation dues aux données partagées (une tâche est dite *prête* dès que toutes les versions des objets partagés auxquelles elle aura accès durant son exécution deviennent valides), que toute tâche créée sera exécutée une et seulement une seule fois et qu'aucune synchronisation ne peut affecter une tâche durant son exécution.

7.4.2 Phases d'exécution

L'exécution d'un programme ATHAPASCAN se décompose en quatre phases :

1. Détection des synchronisations et construction du graphe de flots de données.
2. Calcul d'un ordonnancement du graphe : calcul du placement des tâches sur les différents processeurs et calcul du placement des objets partagés.
3. Diffusion du graphe et de l'ordonnancement aux processeurs impliqués, et génération des communications.
4. Exécution des tâches du graphe.

7.4.3 Construction du graphe de flots de données

L'exécution d'un programme ATHAPASCAN est représentée par un *graphe de flots de données* construit durant l'exécution [45, 46]. Ce graphe permet de représenter non seulement les contraintes de précédences entre les tâches, mais également les types d'accès aux données partagées.

Il est construit à l'aide de deux types de noeuds : les noeuds représentant les tâches et ceux représentant un objet partagé accédé par une ou plusieurs tâches en concurrence. Pour le construire ATHAPASCAN procède ainsi :

- Pour chaque déclaration d'un `Shared` dans le programme, ATHAPASCAN rajoute cette donnée partagée dans le graphe.
- Pour chaque chaque instruction `Fork` rencontrée dans le programme, ATHAPASCAN crée la tâche correspondante et la rajoute dans le graphe en tenant compte de ses contraintes de précédence engendrées par les données partagées que la tâche a comme paramètres.

La figure 7.2 présente le graphe de flots de données associé à la simulation de textiles. Les rectangles, les losanges et les cercles représentent respectivement les accès aux données partagées, les données partagées et les tâches de calculs. Les traits représentent les dépendances.

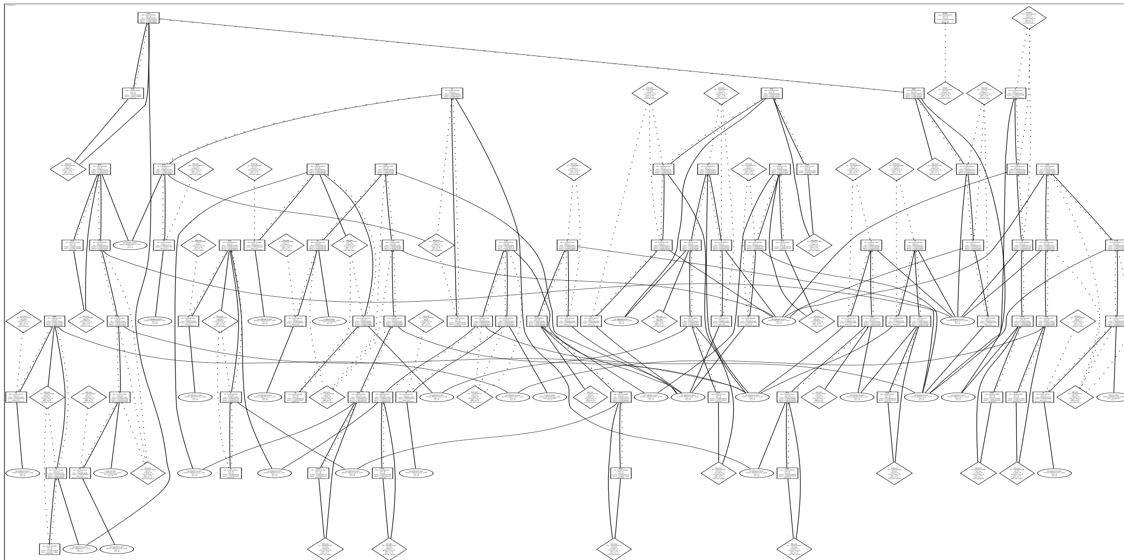


Figure 7.2 *Graphe de flots de données de la simulation de textiles*

A partir de ce graphe, un algorithme d'ordonnancement est appliqué afin de placer au mieux les tâches et les données sur les processeurs grâce notamment aux informations

fournies sur les dépendances des tâches. Ce graphe et cet ordonnancement sont ensuite diffusés à l'ensemble des processeurs impliqués. Puis les communications sont générées entre les processeurs sur lesquels les tâches traitant les données ont été ordonnancées.

7.4.4 Support d'exécution

Pour obtenir une exécution efficace sur l'ensemble fini des ressources que comporte la grappe de machines (processeurs et mémoires inter-connectés par un réseau), le support d'exécution d'ATHAPASCAN [104] doit obligatoirement être capable d'exploiter efficacement au moins deux niveaux de parallélisme : le parallélisme entre plusieurs processeurs partageant une mémoire partagée et le parallélisme existant entre plusieurs noeuds d'une machine multiprocesseurs.

L'emploi de processus légers (ou *threads*) permet une exploitation efficace des ressources de parallélisme présentes dans les machines SMPs. C'est pourquoi l'implantation de l'environnement ATHAPASCAN est basée à la fois sur l'emploi de processus avec leur propre espace d'adressage qui communiquent entre eux et sur l'emploi de processus légers partageant l'espace d'adressage d'un même processus. Au sein d'un noeud, il existe ainsi un ensemble de processus légers qui sont chargés d'exécuter les tâches du programme ATHAPASCAN, ces processus légers communiquant via la mémoire partagée du noeud.

Afin de connaître l'ensemble des tâches prêtes quand un processus léger devient inactif, chaque processus léger doit gérer une liste contenant ces tâches. Une stratégie de *vol de tâches* est ensuite implantée entre les listes de tous les processus légers. La duplication des listes et l'emploi de cette stratégie sont réalisés afin d'éviter d'avoir un goulot d'étranglement au niveau de l'accès à cette liste.

Par ailleurs, chaque processus léger possède sa propre pile. Cette pile sert notamment à stocker la valeur d'un objet partagé devant être accédé par le processus léger. En effet, quand un processus léger exécute une tâche nécessitant une donnée partagée non placée sur ce thread, une tâche de communication est automatiquement créée par ATHAPASCAN afin que ce processus léger en possède une version. Les objets créés par le thread lors de l'exécution du programme ATHAPASCAN sont également stockés dans cette pile, ainsi que les listes des tâches prêtes. L'emploi de cette pile permet une localité des données.

L'ordonnancement local et répartie des tâches sont ainsi basés sur l'ordre de création des tâches : quand un processus léger devient inactif et ainsi prêt à exécuter une tâche, ce processus léger devient un thread *voleur* et commence à rechercher parmi toutes les piles allouées une tâche prête. Deux possibilités s'offrent alors à lui : soit il vole toute la pile associée à la tâche prête, soit il recopie uniquement la tâche (fonction et paramètres associés) volée dans sa propre pile ou dans les piles des autres processus légers. Le fait de stocker ces listes de tâches prêtes dans les piles permet également une identification claire des synchronisations entre un *voleur* et un *volé*. En effet, dans le cas général, si

deux threads exécutent un programme ATHAPASCAN, aucune synchronisation n'a lieu entre ces threads. Les synchronisations n'apparaissent que lors d'un vol ou bien à la fin du programme.

Par ailleurs, le graphe de flots de données étant conservé dans chacune des piles des processus légers, dès la fin de l'exécution d'une tâche il est facile de calculer le noeud d'exécution qui nécessitera la connaissance des données partagées modifiées par cette tâche et ainsi de les transférer via une variante des messages actifs.

7.5 Conclusion

Le parallélisme au sein du langage ATHAPASCAN s'exprime par la création de tâches. Seuls deux mots-clés sont employés pour spécifier les tâches à exécuter en parallèle et les données qui seront partagées par plusieurs processeurs (`Fork` et `Shared`). L'exécution parallèle du programme est basée sur la construction du graphe de flots de données associé à l'application. Ce graphe permet à chaque processeur de connaître les dépendances qui existent entre les tâches dues aux accès aux données partagées. Par ailleurs, afin d'exploiter le plus efficacement possible les ressources disponibles sur une grappe de SMPs, ATHAPASCAN combine un support d'exécution distribuée gérant l'ordonnancement global des tâches parmi l'ensemble des noeuds, avec un support d'exécution SMP gérant l'ordonnancement local des tâches sur chaque processeur.

Parallélisation et couplage de programmes parallèles

Dans la première partie, le problème relatif à la simulation de textiles a été présenté. En résumé, à chaque pas de temps le calcul des états des particules décrivant le tissu doit être effectué. Les mêmes calculs sont ainsi réalisés pour chacune des particules contenues dans le système. Certains de ces calculs peuvent être réalisés sans aucune connaissance concernant les particules voisines de celle considérée (accélération, intégration explicite), tandis que d'autres nécessitent des informations sur ce voisinage (force, intégration implicite).

Lors de la simulation de scènes 3D complexes, les temps de calculs peuvent devenir trop importants pour espérer effectuer des simulations en temps réel. Ce temps de calcul peut être diminué par la parallélisation des algorithmes de simulation permettant l'exécution de l'application sur une machine parallèle de type grappe. Nous avons vu comment d'un point de vue théorique, il était possible de partitionner le problème initial en tâches de calculs plus ou moins dépendantes. Nous avons également présenté l'environnement de programmation parallèle ATHAPASCAN *utilisé pour effectuer cette parallélisation.*

Dans cette dernière partie, nous allons présenter tous les détails de la parallélisation ainsi que les résultats que nous avons obtenus en terme de performances. Nous verrons également comment nous avons couplé notre simulation de textiles en parallèle avec un environnement de réalité virtuelle permettant une visualisation multi-écrans.

Partitionnement 8

Lors de la simulation de scènes 3D complexes, les temps de calculs peuvent devenir trop importants pour espérer effectuer des simulations en temps réel. Ce temps de calcul peut être diminué par la parallélisation des algorithmes de simulation permettant l'exécution de l'application sur une machine parallèle de type grappe. Nous avons vu comment, d'un point de vue théorique, il était possible de partitionner le problème initial en tâches de calculs plus ou moins dépendantes. Nous avons également présenté l'environnement de programmation parallèle ATHAPASCAN *qui va être utilisé pour effectuer notre parallélisation. Cet environnement permet notamment l'obtention d'une application parallèle portable, c'est-à-dire exécutable sur n'importe quel type d'architecture parallèle (machine multiprocesseurs, grappe de processeurs, grappe de multiprocesseurs) et efficace.*

La parallélisation implantée pour la simulation de vêtements est basée sur un partitionnement de l'objet à simuler. Ce chapitre présente ce découpage, ainsi que les tâches qui seront associées aux fragments de données issus du partitionnement.

8.1 Introduction

La simulation de textiles s'exprime sous la forme d'une boucle infinie à l'intérieur de laquelle à chaque pas de temps, les états (positions, vitesses) des particules décrivant le tissu doivent être calculés. Les accélérations sont établies à partir de la loi fondamentale de la dynamique. Le calcul des forces appliquées sur chacune des particules doit donc être effectué au préalable. Ces forces sont essentiellement dues aux ressorts reliant les particules permettant d'émuler un comportement réaliste de leurs interactions, et à des forces extérieures telles que la gravité ou le vent. Le calcul des forces des ressorts pour une par-

ticule donnée nécessite ainsi la connaissance de ses voisins, entraînant des dépendances de données. L'intégration de l'accélération des particules permet ensuite de déterminer leurs vitesses et leurs positions. Deux types de schémas peuvent alors être utilisés : un schéma explicite ou un schéma implicite nécessitant la résolution d'un système linéaire (cf. partie I).

L'algorithme de la simulation de textiles est ainsi irrégulier du point de vue du parallélisme. Le calcul des forces des ressorts appliquées à une particule dépend en effet de ses voisins. De plus, ces dépendances se retrouvent également dans la résolution du système linéaire créé lors de l'emploi d'un schéma d'intégration implicite impliquant des structures de données creuses.

Notre méthode de parallélisation est basée sur un partitionnement du problème initial en sous-tâches de calculs comme l'illustrait la figure 5.1 de la section 5.2.1. Ce chapitre débute par des références bibliographiques sur des travaux effectués sur la parallélisation de la méthode du Gradient Conjugué (GC) utilisée dans notre application lors de l'emploi du schéma d'Euler implicite. L'idée générale de l'algorithme parallèle du GC est notamment explicitée. Ce chapitre se poursuit ensuite par la présentation de deux applications parallèles basées sur une décomposition spatiale qui mettent en évidence l'importance du découpage. Puis nous détaillerons l'élaboration de notre propre partitionnement, pour ensuite présenter les différentes tâches de calculs qu'il sera nécessaire de créer pour effectuer notre simulation parallèle.

8.2 Algorithme parallèle du Gradient Conjugué

L'algorithme du Gradient Conjugué comporte principalement des calculs d'algèbre linéaire de type produits matrice-vecteur ou produits scalaires (cf. 3.5.3, page 43). De plus dans le cadre de la simulation de textiles, la matrice principale de l'algorithme possède une structure creuse. Notre objectif est donc de réussir à paralléliser le plus efficacement possible la méthode du Gradient Conjugué associée à des structures de données creuses.

8.2.1 Références bibliographiques

De nombreux articles relatent des problèmes liés à la gestion de structures de données creuses, ainsi que ceux rencontrés pour effectuer des calculs d'algèbre linéaire en parallèle. Nous invitons donc le lecteur à se référer aux publications que nous allons décrire brièvement ci-après, pour obtenir de plus amples détails.

Meier et Eigenmann [85] publient une méthode de parallélisation de l'algorithme du Gradient Conjugué dédiée à une machine parallèle multiprocesseurs comportant une mémoire hiérarchique (Cedar). La matrice A et les vecteurs sont partitionnés en blocs, puis

sont distribués sur les processeurs. Chaque processeur effectue ensuite les calculs relatifs à ses données. Des points de synchronisation sont rajoutés dans l'algorithme pour effectuer les produits scalaires. En effet, dès qu'un processeur a terminé sa part de calcul pour le produit scalaire, il le signale aux autres. Ensuite, quand tous les processeurs ont terminé, ils peuvent lire les valeurs calculées par les autres processeurs et les accumuler pour obtenir le produit scalaire final. Nous pouvons dès lors noter une redondance de calcul dans cette méthode, puisque tous les processeurs effectuent la même somme finale.

Gallivan *et al.* [49, 47, 48] ont publié des articles relatant notamment des techniques permettant d'implanter en parallèle des méthodes de résolution de systèmes linéaires creux non symétriques. Cette parallélisation se base sur une réorganisation des données de la matrice A permettant l'obtention d'une forme triangulaire supérieure. Une stratégie de placement est alors employée afin de distribuer les données sur les processeurs disponibles. Cette méthode est également dédiée au système parallèle Cedar.

Ujaldon *et al.* [121, 120, 10, 106] utilisent le format HBF pour stocker les valeurs non nulles d'une matrice creuse A. Cette matrice creuse A est au préalable décomposée en plusieurs sous-matrices creuses rectangulaires. Ces sous-matrices peuvent ensuite utiliser la même technique de stockage que pour A. Cette technique de décomposition est employée récursivement pour obtenir autant de sous-matrices que de processeurs. Les index des tableaux sont reconstruits au fur et à mesure afin que chaque processeur possède l'ensemble des données nécessaires à l'accès des sous-matrices locales.

Demmel, Heath et van der Vorst, dans leur rapport de recherche relatif à LAPACK intitulé “Parallel numerical linear algebra” [36], présentent les difficultés liées aux calculs d'algèbre linéaire effectués sur différents types de machines parallèles (mémoire partagée, mémoire distribuée). Ils débutent avec un simple produit matrice-vecteur pour ensuite détailler des algorithmes plus compliqués tels que la méthode du Gradient Conjugué, la factorisation de Choleski, ou encore la méthode de Jacobi pour le calcul des valeurs propres d'une matrice. Ils abordent également quelques techniques de restructuration de données pour gérer des matrices creuses.

D'autre part, dans leur livre intitulé “Parallel and distributed computation”, Bertsekas et Tsitsiklis [19] présentent brièvement des pistes de parallélisation de l'algorithme du Gradient Conjugué, ainsi que Kumar *et al.* [75] dans leur livre “Introduction to parallel computing”.

8.2.2 Algorithme parallèle

Nous discutons brièvement ici des solutions qui existent pour calculer en parallèle les différents produits scalaires et produits matrice-vecteur que comportent l'algorithme du Gradient Conjugué :

$$\alpha \leftarrow r^T r;\ d \leftarrow r + (\frac{\alpha}{\beta})d;\ \beta \leftarrow d^T Ad;\ r \leftarrow r - (\frac{\alpha}{\beta})Ad;\ x \leftarrow x + (\frac{\alpha}{\beta})d;$$

Supposons qu'au début de la $i^{ème}$ itération, les vecteurs de positions, résidus et de directions, soient $x(t), r(t-1), d(t-1)$, aient déjà été calculés. Nous avons alors à évaluer $Ax(t)$, puis le produit scalaire $r(t)^T r(t)$ pour l'obtention du nouveau pas $\alpha(t)$. Ensuite nous devons calculer le produit scalaire $d(t)^T Ad(t)$ pour obtenir le facteur d'erreur $\beta(t)$ et finalement évaluer $Ad(t)$ pour obtenir le nouveau vecteur de résidu. En négligeant les additions de vecteurs et les multiplications de vecteur par un scalaire, les calculs principaux de l'algorithme sont donc deux multiplications matrice-vecteur et deux produits scalaires. De plus, nous pouvons noter que c'est la même matrice A qui est impliquée dans ces deux multiplications matrice-vecteur.

Prenons le cas des architectures parallèles de passages de messages. Si N processeurs sont disponibles, il est alors naturel de laisser le $i^{ème}$ processeur contrôler la $i^{ème}$ composante des vecteurs à calculer, soient $x(t), r(t)$ et $d(t)$. Les produits scalaires de ces vecteurs sont calculés en laissant le $i^{ème}$ processeur calculer le produit entre les $i^{èmes}$ composantes des vecteurs et ensuite les sommes partielles sont accumulées le long des processeurs. Nous avons alors un seul noeud d'accumulation. Puis les valeurs obtenues pour les différents produits scalaires sont diffusées à tous les processeurs.

Nous supposons désormais que chaque processeur possède les valeurs de lignes différentes de la matrice A. Le produit matrice-vecteur $Ax(t)$ est alors calculé en diffusant le vecteur $x(t)$ au préalable à tous les processeurs et ensuite le $i^{ème}$ processeur peut calculer le produits scalaire entre x et la $i^{ème}$ ligne de A. Il existe également une autre possibilité pour effectuer ce calcul. Le $i^{ème}$ processeur peut calculer $[A]_{ji}x_i$ pour chaque j, et cette quantité peut ensuite être propagée à chaque processeur j, la somme partielle étant effectuée au fur et à mesure.

De plus, si moins de N processeurs sont disponibles, la solution est la même sauf qu'il y a alors plus de lignes et plus de composantes de vecteurs qui sont assignées au départ aux processeurs.

D'autre part, il ne faut pas oublier que dans notre cas la matrice A est une matrice creuse. La multiplication de A par n'importe quel vecteur est effectuée plus efficacement en utilisant une structure de donnée appropriée exploitant la non densité de la matrice. La limitation de l'efficacité provient alors généralement du réseau à cause des communications à effectuer entre les processeurs. Il y a alors un compromis à trouver entre le nombre minimal de processeurs à utiliser et le le nombre de composantes à assigner à chaque processeur afin que le nombre de calculs à effectuer sur chaque processeur soit comparable aux communications. Il y a donc toute une phase de calibrage à effectuer afin de trouver la quantification optimale de chacun des paramètres. Celle-ci sera mise en évidence au cours des expérimentations.

De plus amples détails en ce qui concerne nos choix de parallélisation de la méthode du Gradient Conjugué seront fournis au sein du chapitre 9.

8.3 Simulations parallèles

Avant d'expliciter en détails notre propre méthode de parallélisation, il semble important de présenter brièvement quelques autres travaux de recherche se rapprochant de notre travail. A notre connaissance, seules deux autres équipes de recherche travaillent actuellement sur la parallélisation d'algorithmes de simulation de vêtements : Romero *et al.* [105, 107, 108] et Lario *et al.* [76]. Mais ces derniers utilisent des algorithmes de simulation à plusieurs niveaux très différents de ceux que nous utilisons. C'est pourquoi nous ne citons ces travaux qu'à titre indicatif. Par contre nous verrons quelles méthodes Romero *et al.* ont employées pour effectuer leur parallélisation. Un autre travail de recherche a retenu notre attention, celui de Pierre-Eric Bernard [16] dans le cadre de la dynamique moléculaire mettant en évidence les avantages et les inconvénients liés à une parallélisation basée sur un partitionnement de l'espace de simulation.

8.3.1 Simulation de vêtements : Romero *et al.*

Romero *et al.* [105, 107, 108] s'intéressent donc également à la parallélisation des algorithmes de simulation de textiles. Leur machine parallèle cible est une machine parallèle multiprocesseurs (SGI Origin2000). Leur parallélisation est essentiellement basée sur une stratégie de parallélisme de données. La distribution des objets de la scène sur les processeurs est effectuée de façon à répartir équitablement ces objets. Au sein même d'un objet, les éléments (particules, triangles, forces) sont redistribués et réorganisés en utilisant des méthodes de décomposition de domaines. Cette distribution n'est effectuée que durant la phase d'initialisation de l'algorithme, assurant une grande localité des données durant la simulation. Ces données dans certains cas peuvent être répliquées et traitées sur plusieurs processeurs. Dans le cadre de la simulation, ce problème apparaît lors du calcul des forces. Si les voisines d'une particule donnée se trouvent sur d'autres processeurs, celles-ci sont dupliquées afin de se retrouver sur le même processeur. Cela permet notamment d'éviter l'établissement d'une communication ainsi que l'attente de la part d'un processeur des données traitées par un autre. En d'autres termes, ils ont privilégié les redondances de calculs plutôt que les communications entre processeurs. Ensuite les contributions de chacun des processeurs sont accumulées afin d'obtenir le système final.

Leurs performances montrent l'importance de la réorganisation des données. Au départ la matrice creuse intervenant dans l'utilisation d'une méthode d'intégration implicite comporte des éléments non nuls un peu éparpillés, correspondant aux interactions entre particules (figure 4.1, page 52). Puis ces éléments sont réorganisés de façon à diminuer par la suite les communications en augmentant la localité des données. Pour cela deux techniques sont employées : une méthode de distribution récursive et une méthode de réorganisation en bandes. La figure 8.1 montre l'effet de ces réorganisations sur la composition de la matrice creuse.

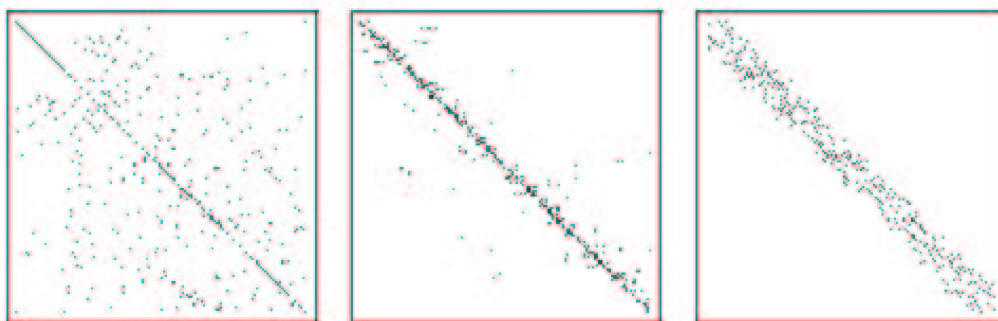


Figure 8.1 *Coefficients de la matrice originale à gauche. Après une réorganisation MRD (Multiple Recursive Distribution) au milieu. Et après une réorganisation en bandes à droite.*

Pour calculer l'ensemble des forces appliquées sur les particules constituant le maillage, l'itération s'effectue sur les forces et non sur les particules [107]. Une force donnée est calculée en accumulant les contributions des forces exercées par les particules voisines. La figure 8.2 illustre l'effet des réorganisations sur ce calcul des forces en terme de localité de données. Les contributions sont ainsi accumulées dans le vecteur `AccForce` et le vecteur `Liste` permet de connaître la liste des particules impliquées dans ce calcul.

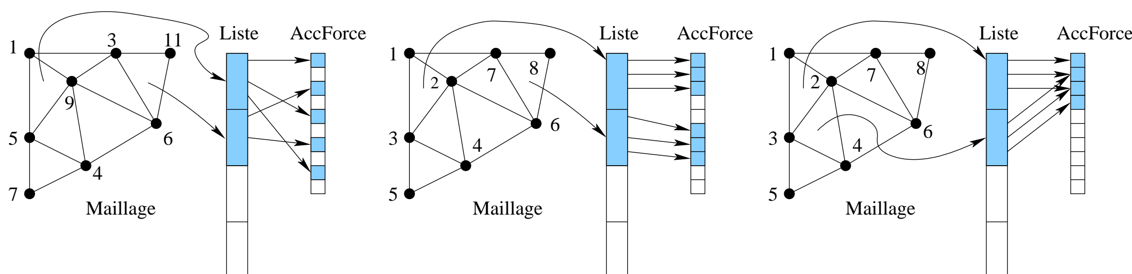


Figure 8.2 *Distribution originale à gauche. Après une réorganisation des particules au milieu. Et après une réorganisation des triangles à droite.*

En ce qui concerne les performances, cette diminution des communications se perçoit très nettement avec l'obtention d'accélérations bien supérieures après une réorganisation des données qu'avec l'utilisation de la matrice originale sans aucune réorganisation [105, 107, 108].

Ce travail de recherche met donc en évidence le fait qu'une réorganisation des données dans le but de minimiser les communications permet de tirer un plus grand profit de la parallélisation effectuée.

8.3.2 Dynamique moléculaire : Bernard *et al.*

Nous reprenons ici les explications de la parallélisation de l'application de dynamique moléculaire élaborée durant le travail de doctorat de Pierre-Eric Bernard [16]. La dynamique moléculaire consiste en la simulation du mouvement d'atomes et de molécules contenus d'un système donné. Pour simuler ces mouvements, les équations de la mécanique classique sont employées. Dans ce modèle, les atomes sont considérés comme des points matériels assortis de données statiques : une masse, une charge électrique et un type (azote, carbone, hydrogène, oxygène, ...). A chaque atome est également associé un vecteur vitesse et une position dans l'espace évoluant au cours du temps.

Le mouvement de ces atomes est régi par l'équation du mouvement de Newton ou par l'équation du mouvement de Langevin suivant le type de simulation souhaité. Pour calculer le mouvement des atomes d'un système, les équations du mouvement sont intégrées de manière itérative. Pour cette intégration, il faut à chaque pas d'intégration calculer les forces qui s'exercent entre les atomes du système. Trois types de forces sont considérées : (1) les forces des interactions non-liées s'exerçant sur chacune des paires d'atomes du système (forces électrostatiques de Coulomb et des forces de Van Der Waals), (2) les forces des interactions géométriques rendant compte des déformations géométriques entre ces atomes, (3) les forces de contraintes permettant de simuler l'influence du milieu extérieur sur le système étudié. Afin de réduire la complexité algorithmique du calcul des forces d'interactions non-liées, la simulation se limite à calculer que celles impliquées entre atomes se trouvant à une distance bornée (méthode du rayon de coupure).

L'algorithme principal de la dynamique moléculaire est ainsi identique à celui de la simulation de vêtements : seul le calcul des forces exercées sur les atomes diffère mais implique les mêmes types de dépendances au niveau du voisinage. Il faut noter que 90% du temps de calcul de la dynamique moléculaire est dû aux forces (1) tandis qu'environ 50% du temps de calcul de la simulation de textiles est passé dans le calcul des forces (2). En plus du voisinage local, le voisinage spatial est également à respecter dans le cadre de la dynamique moléculaire engendrant des communications entre processeurs.

C'est pourquoi pour paralléliser ce type d'application de dynamique moléculaire, la meilleure technique en terme de communications consiste à découper l'espace géométrique [16, 50, 17, 18]. Chacun des processeurs se charge ensuite d'une région donnée de l'espace de simulation. Les atomes ne sont alors pas liés à un processeur, mais peuvent changer de processeur au cours de la simulation. L'espace de simulation est ainsi divisé en boites cubiques, dont la taille dépend du rayon de coupure. Ainsi les atomes voisins d'un atome donné se trouvent soit dans la boîte de l'atome considéré soit dans une des 26 ($3^3 - 1$) boîtes voisines. La figure 8.3 montre une vue en trois dimensions d'un tel découpage de l'espace.

Si une boîte est placée par processeur, chacun des processeurs échange la position des atomes de sa boîte avec les 26 processeurs possédant les boîtes voisines. Après cette communication, chacun des processeurs possède les coordonnées des atomes de sa boîte

et celles des atomes des boîtes voisines. Il peut ainsi calculer les forces d'interactions non-liées et les forces d'interaction géométriques qui s'exercent sur les atomes de sa boîte. Pour un ensemble de n atomes et p processeurs, l'algorithme est en $\mathcal{O}(\frac{n}{p})$ pour les échanges de coordonnées des atomes entre les processeurs contenant les boîtes voisines, en $\mathcal{O}(\frac{n}{p})$ pour le calcul des forces s'exerçant sur les atomes de la boîte locale, et en $\mathcal{O}(\frac{n}{p})$ pour l'intégration de l'équation du mouvement pour les atomes de la boîte locale.

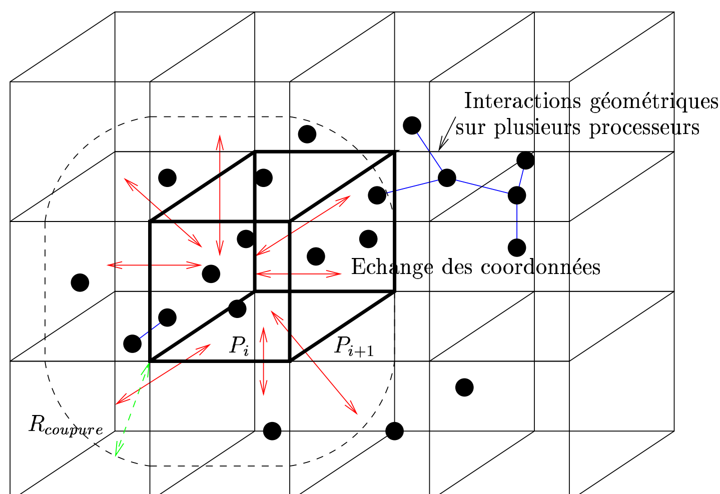


Figure 8.3 *Découpage spatial de l'espace de simulation dans le cadre d'une application de dynamique moléculaire*

L'inconvénient majeur de cette méthode résulte de la variabilité du nombre d'atomes présents dans les boîtes. Le déséquilibre du nombre d'atomes par processeur entraîne en effet un déséquilibre de la charge de calcul. Il est alors impératif d'avoir recours à une stratégie permettant de rééquilibrer les calculs sur les processeurs au cours de la simulation [16, 17].

8.4 Partitionnement

Revenons à la parallélisation de notre application de textiles. La première étape dans l'élaboration d'un algorithme parallèle consiste à partitionner le problème initial en sous problèmes. Nous avons vu dans la section 5.2.2 (page 64) qu'il existait deux types de décompositions : la décomposition de l'espace et la décomposition fonctionnelle. C'est-à-dire que pour diviser le problème initial en sous-tâches, soit nous décomposons les données en sous-ensembles sur lesquels les mêmes calculs seront établis en concurrence, soit nous commençons par décomposer le problème en tâches de calculs, puis à partir de ces tâches nous analysons quelles données seront nécessaires à leur élaboration.

Nous allons voir que notre méthode de parallélisation est une combinaison de ces deux techniques de partitionnement.

8.4.1 Décomposition de l'espace objet

La première étape lors d'une décomposition spatiale consiste à partitionner les données en fragments de taille plus petite. Dans le cadre de la simulation de textiles ces données correspondent aux particules (forces, positions, vitesses et accélérations). L'ensemble des particules contenues dans le système est ainsi fractionné en sous-ensembles.

Cette découpe est en fait la base de notre parallélisation. Son choix va en effet fixer le graphe de calcul de l'algorithme en définissant les tâches parallèles et les données partagées. A partir de cette définition ATHAPASCAN pourra créer le graphe de flots de données associé à notre application. Par conséquent, le choix de cette découpe doit être guidé par les performances de l'exécution et en particulier par le fait que ce graphe ne soit pas remis en question durant toute la simulation contrairement au cas de l'application de dynamique moléculaire.

En effet, dans le cas de la dynamique moléculaire ce graphe (données + tâches) dépend essentiellement de la position des atomes : le calcul des forces et l'équilibrage de charge dépendant des voisinages de chacun des atomes. La meilleure heuristique consiste alors à découper l'espace de simulation en boîtes, puis à fixer des boîtes par processeur pour ensuite laisser se déplacer les atomes parmi ces boîtes durant un certain nombre d'itérations jusqu'à une étape de redistribution remettant en cause le placement initial.

Tandis que dans le cas de la simulation de textiles, le calcul des forces et l'équilibrage de charge ne dépendent à priori que des liaisons entre particules. Or ces liaisons ne changeant pas au cours du temps, il n'y a aucune raison de changer le graphe initialement créé. Par conséquent, le fractionnement réalisé une seule fois durant la phase d'initialisation, consiste à découper le tissu en sous-ensembles : chaque particule est au départ attribuée à un sous-ensemble et aucune particule ne change de sous-ensemble au cours de la simulation.

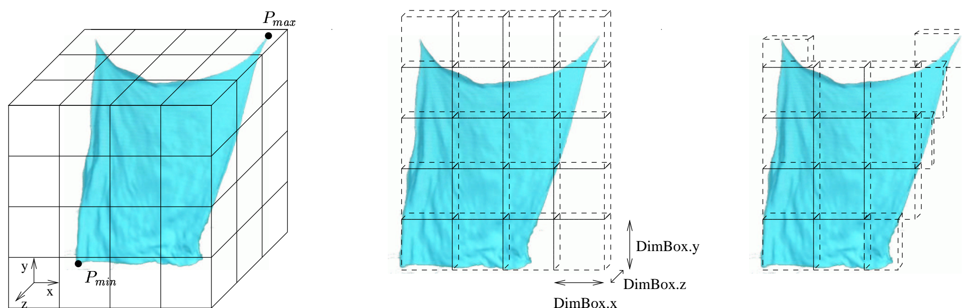


Figure 8.4 *Décomposition du tissu en sous ensembles de particules de dimension* `DimBox`

La figure 8.4 illustre le découpage de la simulation de textiles. Les positions minimale et maximale de l'ensemble des particules sont tout d'abord déterminées. Puis les particules sont affectées dans les boîtes dont la dimension `DimBox` est donnée en entrée de l'application, la première boîte commençant par la particule de position minimale.

En terme de structures de données, si l'ensemble des données relatives aux particules (forces, positions, vitesses, accélérations, masses) est stockée dans des tableaux (`Force[]`, `Position[]`, `Vitesse[]`, `Accel[]`, `Masse[]`, ...), le partitionnement consistera à découper ces tableaux en blocs. Ce découpage est illustré par la figure 8.5.

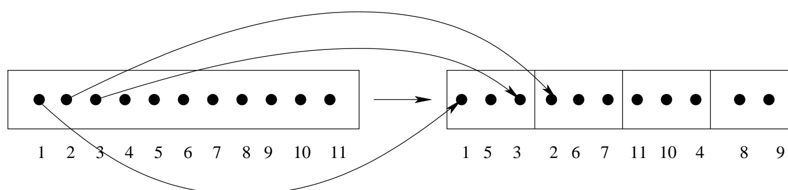


Figure 8.5 *Structures de données reflétant le partitionnement de l'objet simulé : à gauche le tableau contient l'ensemble des particules, à droite ce tableau a été découpé en blocs contenant des sous-ensembles de particules*

Ces blocs de données représenteront donc les objets partagés de notre application parallèle permettant par la suite la définition des tâches de calculs.

Optimisation de la découpe

L'avantage principal d'effectuer le partitionnement dans l'espace de l'objet et non dans celui de la simulation comme dans le cadre de la dynamique moléculaire, réside dans l'obtention de blocs englobant le même nombre de particules. Ceci permet notamment une répartition équilibrée de la charge de calculs sur les processeurs. Mais même si dans le cas d'un maillage régulier il est très facile de décomposer le tissu en blocs de même taille, cela devient plus critique dans le cas général où le maillage représentant l'objet peut être irrégulier.

Il est alors nécessaire d'avoir recours à des techniques bien établies pour effectuer ce découpage de manière optimale. Il y a en réalité deux alternatives possibles : soit ce découpage optimal est effectué dès le départ, soit un découpage trivial est dans un premier temps élaboré pour ensuite effectuer une réorganisation des particules au sein des blocs permettant à terme l'obtention d'un découpage optimal. La simulation parallèle de Romero *et al.*, présentée dans la section 8.3.1, utilise notamment cette seconde technique de réorganisation des particules de façon à regrouper les éléments non nuls de la matrice initialement éparpillés et ainsi de diminuer la largeur de la bande de la matrice.

Des bibliothèques de partitionnement peuvent être utilisées pour effectuer cette fragmentation de l'ensemble des particules. La bibliothèque Scotch [94, 93] par exemple offre ces deux alternatives de réorganisation ou de partitionnement effectués en fonction de certains critères que l'utilisateur doit définir. La bibliothèque Metis [69, 70, 110] permet également de partitionner des graphes.

L'emploi de certaines courbes mathématiques permet également d'établir une organisation en une dimension des particules initialement définies en deux dimensions via leurs positions. Cette indexation est établie avec le souci de préserver la proximité spatiale et temporelle des éléments. En effet ces courbes mathématiques permettent de parcourir un chemin sur une surface bidimensionnelle de façon à rejoindre tous les points grâce aux propriétés suivantes : (1) la courbe continue passe une seule fois à travers tous les points, (2) les points voisins dans la courbe le sont dans l'espace, (3) la courbe constitue une transformée entre elle même et l'espace à n dimensions.

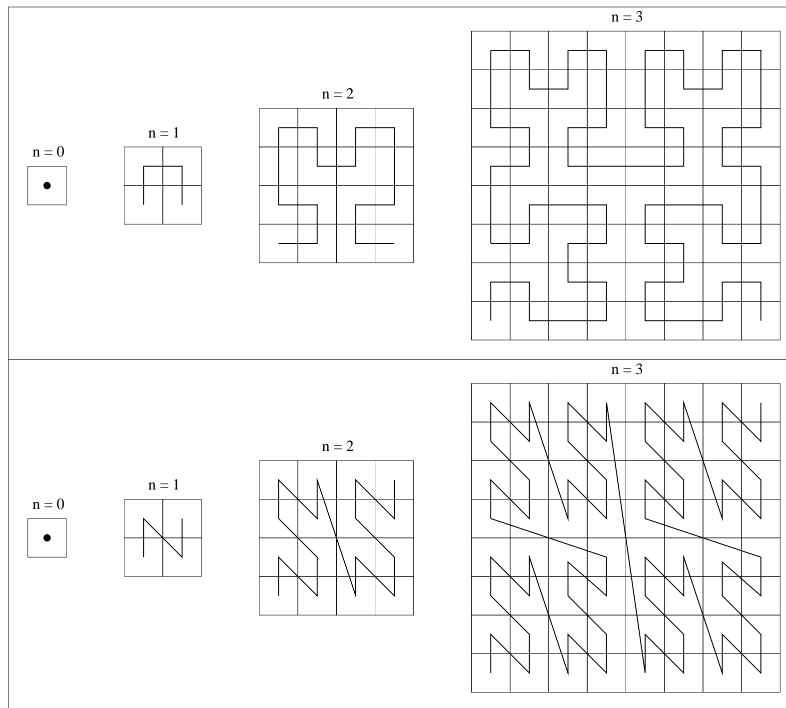


Figure 8.6 *Courbes récursives de Hilbert (en haut) et de Peano (en bas) pour* $n = 0, ..., 3$

Les points sont donc stockés dans l'ordre déterminé par le chemin. L'avantage principal de cette méthode tient à ce que les points proches dans l'espace se retrouvent proches dans la structure de données. De plus l'ordre de parcourt des points augmente la localité temporelle des accès aux données : les derniers points du chemin étant ceux utiles au cal-

cul. Ceci permet notamment de réduire les temps d'accès mémoire et bien entendu dans notre cas, de regrouper au sein d'un même bloc des particules voisines. Les deux types de courbes les plus fréquemment utilisées sont la courbe de Hilbert, illustrée en haut de la figure 8.6 et la courbe de Peano représentée en bas de la même figure. La courbe de Peano permet l'obtention d'une structure stable mais où les points consécutifs ne sont pas toujours adjacents. Par contre le courbe de Hilbert donne lieu à des structures instables mais où les points consécutifs sont toujours adjacents. L'analyse de ces deux courbes a donné lieu à l'élaboration de nombreuses variantes [9, 138] permettant notamment dans un cadre plus général, la diminution du temps de recherche dans des fichiers ou encore la diminution des défauts de cache mémoire. Ce domaine de recherche est toujours d'actualité.

Cette analyse a été faite après les premières implantations de notre application et n'est donc actuellement pas intégrée au sein de notre simulation. Mais elle laisse entrevoir de nouvelles optimisations possibles pour l'élaboration de la découpe.

Optimisation des communications engendrées par la découpe

Nous verrons par la suite de manière détaillée que des communications de données vont avoir lieu entre les tâches présentant des interactions. Ces tâches apparaîtront notamment lors du calcul des interactions entre particules voisines. Si ces particules se trouvent sur des blocs différents, eux-mêmes placés sur des processeurs distincts, le processeur effectuant ce calcul sur un des deux blocs aura besoin des informations contenues dans l'autre bloc, et de là nécessitera une communication.

Considérons le découpage du tissu en deux dimensions. Si nous supposons que chacun des blocs du découpage se trouve sur un processeur différent, les calculs relatifs à un bloc donné nécessitent avec ce partitionnement les communications de tous ses blocs voisins. Ces communications sont illustrées par la figure 8.7.

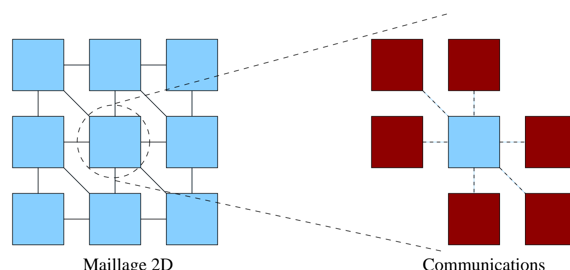


Figure 8.7 *Communications issues du partitionnement du tissu en blocs de particules : cas où chacun des blocs se trouve sur un processeur différent*

Or ces calculs ne nécessitent que des informations sur les particules voisines de ce bloc et non sur l'ensemble des particules des blocs voisins. Il est alors possible de diminuer facilement ces communications en faisant en sorte que seules les frontières des blocs soient communiquées et non l'ensemble du bloc. Les blocs sont alors décomposés en sous blocs comprenant un bloc central et des blocs frontaliers. La figure 8.8 met en évidence ces communications et l'ensemble de ces blocs.

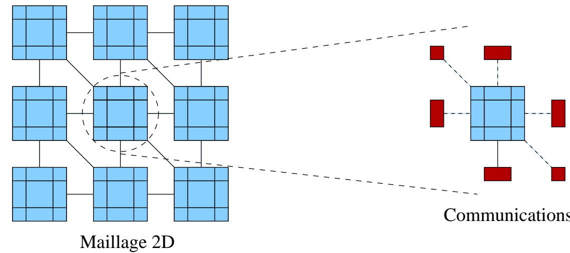


Figure 8.8 *Optimisation des communications issues du partitionnement : seules les frontières des blocs sont transmises aux processeurs ayant les blocs voisins*

Complexité en mémoire

Supposons que nous ayons à notre disposition p processeurs pour réaliser une simulation comprenant n particules partitionnées en k blocs. Il y a ainsi $\frac{n}{k}$ particules par blocs et $\frac{k}{p}$ blocs de taille $\sqrt{\frac{n}{k}} \times \sqrt{\frac{n}{k}}$ par processeur. Considérons le cas au pire c'est-à-dire quand il y a autant de blocs que de processeurs soit seulement 1 bloc par processeur.

Par ailleurs, chaque bloc dispose de 6 voisins et ainsi à partir de ces informations nous pouvons facilement en déduire qu'il y a 6 boîtes voisines pour chaque processeur et donc 6 processeurs voisins pour chaque processeur. Le volume de communication par processeur est donc de $6 \times$ taille d'un bloc communiqué et le volume de communication global est alors de $6 \times$ taille d'un bloc communiqué$\times$ $(p-1)$.

Si nous considérons le premier cas où l'ensemble du bloc est communiqué soit $\frac{n}{k}$ éléments, le volume de communication par processeur est alors de $6 \times \frac{n}{k}$ et le volume global de communication est de $6 \times \frac{n}{k} \times (p-1)$. Dans le deuxième cas seules les frontières des blocs sont communiquées soit $\sqrt{\frac{n}{k}}$ éléments pour les processeurs frontaliers et 1 élément pour les processeurs diagonaux. Le volume de communication par processeur est alors de $4\sqrt{\frac{n}{k}} + 2$ et le volume global de communication est de $(4\sqrt{\frac{n}{k}} + 2) \times (p-1)$.

Nous en déduisons que dans le premier cas le volume de communication par processeur est en $\mathcal{O}(6\frac{n}{k})$ et le volume global en $\mathcal{O}(6n)$, et que dans le second le volume de communication par processeur est en $\mathcal{O}(4\sqrt{\frac{n}{k}})$ et le volume global en $\mathcal{O}(4k\sqrt{\frac{n}{k}})$. La figure 8.9 illustre l'ensemble de ces informations.

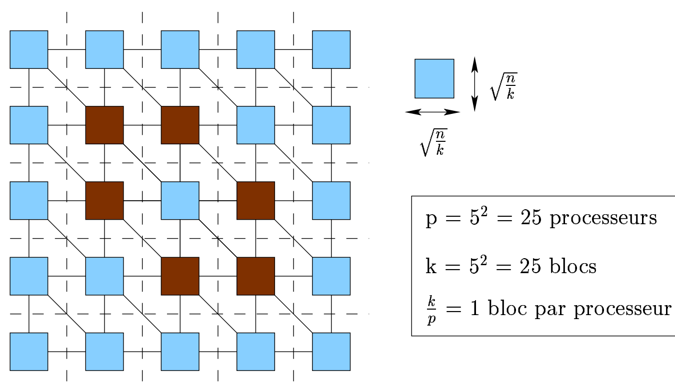


Figure 8.9 *Simulation de n particules sur p processeurs avec un partitionnement en une grille de k blocs*

Le tableau 8.1 résume les complexité en mémoire d'une simulation parallèle sur p processeurs de n particules partitionnées en k blocs.

	Cas 1	Cas 2
Volume de communication par processeur	$\mathcal{O}(\frac{n}{k}))$	$\mathcal{O}(\sqrt{\frac{n}{k}})$
Volume global de communication	$\mathcal{O}(n)$	$\mathcal{O}(k\sqrt{\frac{n}{k}})$

TAB. 8.1 *Comparaison de la complexité en communication dans les deux cas*

Ces complexités mettent ainsi en évidence que plus le bloc de données est important plus le gain en communications obtenu par le transfert des seules frontières l'est aussi. Cette propriété caractérise la plus grande différence qui existe entre notre application de textiles et l'application de dynamique moléculaire. C'est pourquoi le choix de la granularité, dans notre cas la taille des blocs de données, nécessite un compromis entre le volume de communication et le volume des calculs effectués au sein d'une tâche parallèle. Mais ce gain en communications dans le second cas doit être modéré par le coût d'un nombre plus important de tâches à gérer. Les expérimentations mettront en évidence cette quantification à mettre en place.

8.4.2 Tâches associées aux fragments de données

Pour chacune des particules du système, il faut calculer les forces qui lui sont exercées, son accélération, sa vitesse et sa position. Ce sont les quatre types de calculs mis en évidence par le découpage fonctionnel. Une fois la décomposition des données en blocs de particules effectuée, quatre types de tâches de calculs vont donc être créées :

1. le calcul des forces d'interaction entre deux blocs de particules,
2. le calcul des accélérations des particules d'un bloc,
3. le calcul des positions des particules d'un bloc,
4. et le calcul des vitesses des particules d'un bloc.

Ces tâches seront ensuite exécutées en concurrence pour chacun des blocs. La complexité en terme de calculs est ainsi fortement diminuée. En séquentiel la complexité pour un calcul donné est en effet en $\mathcal{O}(n)$ avec n le nombre de particules. En parallèle sur p processeurs avec k blocs, comme le coût pour une boîte est en $\mathcal{O}(\frac{n}{k})$ et qu'il y a $\frac{k}{p}$ boîtes par processeur, la complexité n'est plus qu'en $\mathcal{O}(\frac{n}{p})$.

8.5 Conclusion

La parallélisation de notre simulation de textiles est basée sur un découpage de l'objet en sous-ensembles de particules. Des tâches traitant ces fragments de données vont ensuite être créées et exécutées en parallèle via l'environnement de programmation parallèle ATHAPASCAN. La taille des blocs de données définit alors la granularité de l'algorithme parallèle. Le chapitre suivant présente en détails la création de ces tâches de calculs, ainsi que leur ordonnancement en vue d'optimiser leur exécution sur la grappe de machines.

Création et exécution des tâches 9

Dans le chapitre précédent, les décompositions fonctionnelle et spatiale de la simulation de textiles ont été abordées. L'interface de programmation de l'environnement ATHAPASCAN *a également fait l'objet d'un chapitre précédent. Nous pouvons dès lors exposer plus en détails la création des tâches de calculs parallèles spécifiques à notre application. Ce chapitre débutera ainsi par la présentation de ces tâches, puis nous verrons comment elles sont distribuées et exécutées au sein de la grappe de machines. Nous analyserons notamment les contraintes de précédence qui existent entre elles grâce au graphe de flots de données associé à l'application.*

9.1 Introduction

Le comportement du tissu est reproduit par le mouvement des particules contenues dans le système. A chaque itération il est nécessaire de calculer les états de ces particules impliquant notamment le calcul des forces qui leurs sont appliquées. Le chapitre précédent a permis de mettre en évidence les deux aspects de notre parallélisation. La simulation de textiles est d'une part décomposée en différentes étapes de calculs correspondants aux calculs des états et forces des particules. D'autre part l'objet de la simulation est fragmenté en plusieurs sous ensembles de particules. Chacune des étapes de calculs est ainsi décomposée en tâches de calculs, ces tâches effectuant le traitement relatif aux particules contenues dans un seul bloc.

Nous allons détailler dans ce chapitre ces tâches de calculs. Nous verrons notamment quelles sont leurs contraintes de dépendances et ce qu'elles impliquent en terme de parallélisme.

9.2 Création des tâches

La parallélisation de notre simulation de textiles est basée sur un partitionnement de l'objet. L'ensemble des particules contenues dans le système servant à décrire le tissu est ainsi fragmenté en sous-ensembles. Les structures de données employées reflètent naturellement ce découpage.

9.2.1 Représentation des données

Le tableau 9.1 récapitule l'ensemble des structures de données stockant les propriétés des particules présentes dans l'application. Suite au partitionnement du textile en sous-ensembles de particules, ces structures sont également décomposées en blocs (cf. figure 8.5). Par ailleurs, afin de rendre plus lisibles les graphes de flots de données qui seront présentés tout au long de ce chapitre, nous utiliserons des termes raccourcis à la place des termes exactes. Ces termes sont également définis dans le tableau 9.1.

Type de données	**Nom de la structure**	**Termes raccourcis**
Forces	`Force[]`	force
Positions	`Position[]`	x
Vitesses	`Vitesse[]`	x'
Accélérations	`Accel[]`	x"
Masse	`Masse[]`	m
Contributions des forces (éléments diagonaux)	`Matrice_df_dx_diag[]`	**df/dx**
Contributions des forces (éléments non diagonaux)	`Matrice_df_dx[][]`	df/dx
Vecteur b du système Ax = b	`B[]`	b
Vecteur solution du GC	`X[]`	Δv

TAB. 9.1 *Structures de données de la simulation de textiles : type de données, nom des structures et termes raccourcis employés notamment dans les graphes de flots de données*

La granularité de l'algorithme parallèle implanté avec ATHAPASCAN est définie par la taille des données partagées mises en paramètres des tâches de calculs. Dans notre cas, les calculs sont établis pour les les blocs de particules, ces blocs étant les objets partagés de la simulation parallèle. Le nombre de particules contenues dans un bloc représente par conséquent le grain de notre application parallèle.

A ces structures il faut rajouter une table de hachage nommée `list`. En effet, dans le cas de certaines tâches de calcul, nous verrons qu'il est nécessaire de connaître l'ensemble

des paires de blocs ayant des interactions, et pour une paire de blocs la liste des particules effectivement en interactions.

Afin de ne pas rechercher l'ensemble de ces interactions à chaque itération, une table de hachage est donc créée durant l'initialisation permettant l'enregistrement une fois pour toute de ces interactions. Ses clés correspondent à des paires de blocs. Par exemple admettons qu'il existe des interactions entre les particules du bloc i et celles du bloc j_1 : la clé ($bloc_i$, $bloc_{j1}$) sera par conséquent créée dans la table de hachage. Puis pour une clé donnée, cette table stocke la liste des particules ayant des interactions entre les deux blocs de la clé (appelée `list_part`), ainsi que des pointeurs vers les blocs de données nécessaires aux calculs considérés.

Nous allons désormais détailler chacune des tâches implantées avec ATHAPASCAN dont les algorithmes séquentiels avaient été exposés au sein du chapitre 4.

9.2.2 Calcul des forces

Les forces exercées sur les particules du système sont essentiellement dues aux ressorts les reliant entre elles. Pour une particule donnée, la force qui lui est appliquée correspond à la somme des interactions qu'elle a avec chacune de ses particules voisines. L'algorithme séquentiel de ce calcul est notamment détaillé dans la section 4.2.1 du chapitre 4.

Pour paralléliser ce calcul, nous créons la tâche `Force` (algorithme 12, lignes 1-14), qui va traiter l'ensemble des interactions qui existent entre les particules de deux blocs donnés. Cette tâche ne sera bien entendu créée que pour les paires de blocs ayant des interactions, c'est-à-dire pour lesquels des particules sont reliées par un ressort (lignes 16-18). Pour cela, dans la phase d'initialisation de l'application (cf. section 9.2.1), une table de hachage est créée enregistrant ces interactions entre paires de blocs (`list`, ligne 16). Les clés de cette table correspondent à des paires de blocs, et les données qui sont stockées pour une clé donnée sont la liste des particules ayant des interactions entre ces deux blocs (`list_part`, ligne 18) et des pointeurs vers les blocs de données nécessaires aux calculs des interactions (ligne 18) : les éléments des structures `Position[]`, `Vitesse[]` et `Force[]`.

Prenons l'exemple où il existe des interactions entre les particules du bloc i et celles du bloc j_1 et entre le bloc i et le bloc j_2. Notre objectif est de réussir à calculer les forces exercées sur les particules contenues dans le bloc i. Si nous considérons l'algorithme 12, nous pouvons voir que deux tâches de calculs `Force` vont être créées (ligne 18), l'une avec les données correspondants aux blocs i et j_1, et l'autre avec celles relatives aux blocs i et j_2. Ces deux tâches, précédées du mot-clé `Fork`, seront par la suite exécutées en parallèle par ATHAPASCAN, c'est-à-dire qu'elles pourront être accomplies au même moment par des processeurs ou processus distincts.

Algorithme 12 Calcul parallèle des forces exercées sur les particules

```
1 : struct Force {
2 :   void operator() (a1::Shared_cw< Cumul, Vect3D > f1,
                      a1::Shared_cw< Cumul, Vect3D > f2,
                      a1::Shared_r< Vect3D > v1, a1::Shared_r< Vect3D > v2
                      a1::Shared_r< Vect3D > p1, a1::Shared_r< Vect3D > p2,
                      a1::Shared_r< VectInter> list_part) {
3 :     // Parcours de la liste des particules en interactions
4 :     for(int i=0 ; i<list_part.size() ; ++i) {
5 :       // Interaction entre list_part[i].part1 et list_part[i].part2 : Inter
6 :       // Accumulation des interactions pour chaque particule des 2 blocs
7 :       tmpvectf1[part1] += Inter ;
8 :       tmpvectf2[part2] -= Inter ;
9 :     }
10 :     // Mise a jour des variables partagees
11 :     f1.cumul(tmpvectf1) ;
12 :     f2.cumul(tmpvectf2) ;
13 :   }
14 : } ;

15 : void Calcul_Force(Force[], Position[], Vitesse[], list) {
16 :   // Liste des interactions entre les blocs : list
17 :   while (list.begin() != list.end()) {
18 :     a1::Fork< Force > () (list.f1, list.f2, list.v1, list.v2,
                              list.p1, list.p2, list.list_part) ;
19 :     list++ ;
20 :   }
21 : }
```

A l'intérieur de chacune de ces tâches, les interactions entre les particules du bloc i et celles du bloc j_1 ou j_2 vont être calculées. Ces interactions vont être cumulées dans des tableaux temporaires (lignes 7-8) : pour une particule donnée d'un des blocs, ses interactions avec toutes ses voisines contenues dans l'autre bloc vont être additionnées. Pour cela nous utilisons la liste `liste_part` listant toutes les paires de particules connectées entre ces deux blocs. Enfin, à la fin de la tâche, c'est-à-dire quand toutes les interactions ont été cumulées, les blocs des forces peuvent être mis à jour à partir des variables temporaires (lignes 11-12).

Rappelons que les blocs de données sont des variables partagées au sens ATHAPASCAN, c'est-à-dire que ce sont des données partagées entre les processeurs de la grappe. Il est important de remémorer que ces variables ne peuvent être modifiées qu'au sein

d'une tâche ATHAPASCAN, et qu'il existe différents types d'accès à ces données (lecture, écriture, lecture/écriture ou écriture cumulée).

Dans la tâche `Force`, les blocs relatifs aux positions et vitesses ne devant pas être modifiés, sont pris en lecture seule (`Shared_r`). Par contre les blocs relatifs aux forces sont pris en écriture cumulée (`Shared_cw`) (ligne 2). Cela signifie que ces blocs peuvent être mis à jour au même instant par des processeurs différents, mais que cette modification ne peut être effectuée que selon une fonction précise qui est ici la fonction `Cumul` (ligne 2). Cette fonction doit impérativement être commutative et associative puisque l'ordre des modifications, effectuées par plusieurs processeurs, n'est pas déterminé. Dans notre cas cette fonction va, pour chacun des éléments du bloc, additionner son contenu avec l'élément correspondant de la variable temporaire (ligne 11-12). C'est-à-dire que les deux tâches `Force` créées ont le droit de modifier au même instant le $i^{ème}$ bloc des forces en lui ajoutant les nouvelles interactions calculées. Cette accumulation est illustrée par la figure 9.1.

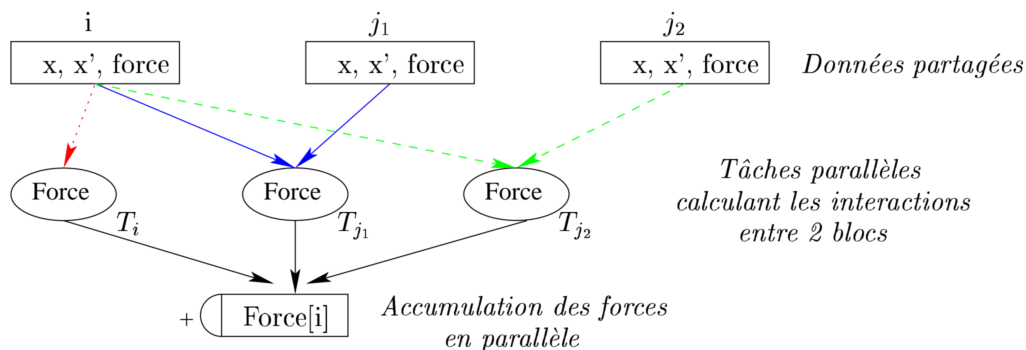


Figure 9.1 *Graphe de flots de données associé aux calculs des forces relatives au bloc i. Les particules contenues dans le $i^{ème}$ bloc ont des interactions avec celles disposées dans les blocs j_1 et j_2 (tâches de calcul T_{j_1} et T_{j_2}), auxquelles il faut rajouter les interactions locales au bloc i (tâche de calcul T_i).*

Mais à ces interactions, il ne faut pas oublier d'y ajouter les interactions locales au bloc i pour obtenir les forces exercées sur les particules de ce bloc. Les interactions locales à un bloc sont calculées de la même manière que celles entre deux blocs distincts. Une clé ($bloc_i$, $bloc_i$) est créée dans la table de hachage, provoquant par la suite l'exécution d'une tâche parallèle `Force` calculant les interactions locales au bloc i.

En résumé, le calcul des forces relatives à un bloc donné nécessite la connaissance des positions et vitesses des particules voisines contenues dans les blocs adjacents. Les tâches parallèles `Force` présentent donc des dépendances de données.

9.2.3 Calcul des accélérations

Les accélérations des particules contenues dans le système sont calculées en appliquant la loi fondamentale de la dynamique. Pour une particule donnée, son accélération correspond au rapport entre les forces qui lui sont appliquées et sa masse qui reste inchangée tout au long de la simulation. L'algorithme séquentiel de ce calcul est fourni dans la section 4.2.2 du chapitre 4.

Pour paralléliser ce calcul, la tâche `Accel` est créée (algorithme 13, lignes 1-6). Cette tâche va effectuer le calcul des accélérations de l'ensemble des particules contenues dans un bloc. Il y aura ainsi autant de tâches parallèles `Accel` créées que de blocs (ligne 8-9).

Algorithme 13 Calcul parallèle des accélérations des particules

```
1 :   struct Accel {
2 :     void operator() (a1::Shared_r< Vect3D > f, a1::Shared_r< Vect > m,
                         a1::Shared_r_w< Vect3D > a) {
3 :          for(int i=0 ; i<a.access().size() ; ++i)
4 :              a.access()[i] = 1/m.read()[i] * f.read()[i] ;
5 :        }
6 :   } ;

7 :   void Calcul_Accel(Force[], Masse[], Accel[], Nb_Blocs) {
8 :        for(int i=0 ; i<Nb_Blocs ; ++i)
9 :             a1::Fork< Accel > () (Force[i], Masse[i], Accel[i]) ;
10 :  }
```

Par exemple, pour effectuer le calcul des accélérations des particules du bloc i, une tâche parallèle `Accel` est instantiée, prenant en paramètres en lecture seule (`Shared_r`) les blocs relatifs aux masses et aux forces des particules du bloc i (ligne 2) et en lecture/écriture (`Shared_r_w`) le bloc des accélérations. Puis pour chacune des particules de ce bloc (ligne 3), le rapport entre sa force et sa masse est effectué pour obtenir sa nouvelle accélération (ligne 4).

Les blocs relatifs aux forces et aux masses étant pris en lecture seule, et aucune modification sur ces données ne devant être effectuée, la méthode `read()` est employée pour lire leurs éléments (ligne 4). Par contre l'accès aux données du bloc des accélérations est effectué via la méthode `access()`, utilisée pour les paramètres pris en lecture/écriture (ligne 4). Le graphe de flots de données associé à ce calcul est représenté par la figure 9.2.

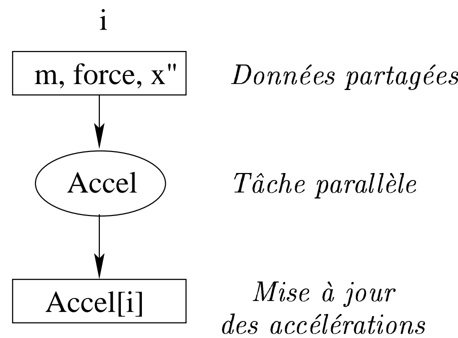


Figure 9.2 *Graphe de flots de données associé aux calculs des accélérations des particules contenues dans le bloc i. Pour calculer leurs accélérations, il suffit de connaître leurs forces, ainsi que leurs masses. Aucune information relative aux autres blocs n'est nécessaire.*

En conclusion, cette tâche de calcul ne nécessite que des connaissances relatives à son propre bloc. Elle est ainsi totalement indépendante en terme de données des autres blocs.

9.2.4 Schémas d'intégration explicites

Après l'obtention des accélérations de chacune des particules, il est dès lors possible de calculer les vitesses et les positions de ces particules. Au sein du chapitre 3, nous avons présenté en détails les trois méthodes d'intégration implantées dans notre application. Deux d'entres elles (Euler explicite et Stöermer-Verlet/leapfrog) sont des méthodes explicites dont les schémas d'intégration sont assez similaires d'un point de vue algorithmique. L'algorithme séquentiel relatif à ces deux schémas est fourni dans la section 4.2.3 du chapitre 4.

Pour paralléliser ce calcul, la tâche parallèle `Integr_Expl` est instantiée (algorithme 14, lignes 1-8). Cette tâche va ainsi effectuer le calcul des positions et vitesses de l'ensemble des particules contenues dans un bloc en utilisant un schéma explicite. De la même manière que pour la tâche calculant les accélérations, il y aura autant de tâches parallèles `Integr_Expl` créées que de blocs (lignes 10-11).

Reprenons en exemple le cas du traitement du bloc i. Une tâche parallèle `Integr_Expl` est créée avec en paramètres en lecture seule (`Shared_r`) le bloc contenant les accélérations des particules de ce bloc, et en lecture/écriture (`Shared_r_w`) les blocs relatifs aux positions et vitesses de ces particules (ligne 2). La tâche effectue ensuite pour chacune des particules du sous ensemble (ligne 3), le calcul de leurs positions (ligne 4) et de leurs vitesses (ligne 5) en employant un schéma d'intégration explicite.

Algorithme 14 Intégration en parallèle via un schéma explicite

```
1 :  struct Integr_Expl {
2 :    void operator() (a1::Shared_r_w< Vect3D > p, a1::Shared_r_w< Vect3D > v,
                       a1::Shared_r< Vect3D > a, double h) {
3 :         for(int i=0 ; i<p.access().size() ; ++i) {
4 :            v.access()[i] = v.access()[i] + h * a.read()[i] ;
5 :            p.access()[i] = p.access()[i] + h * v.access()[i] ;
6 :         }
7 :      }
8 :  } ;

9 :  void Integration_Expl(Position[], Vitesse[], Accel[], h, Nb_Blocs) {
10 :    for(int i=0 ; i<Nb_Blocs ; ++i)
11 :       a1::Fork< Integr_Expl > () (Position[i], Vitesse[i], Accel[i], h) ;
12 :  }
```

Le bloc relatif aux accélérations étant pris en lecture, ses données sont accédées via la méthode `read()` (ligne 4). La méthode `access()` est par contre employée pour les blocs des positions et vitesses dont les données, prises en lecture/écriture, doivent être modifiées (lignes 4-5). Le graphe de flots de données associé à ces calculs est représenté par la figure 9.3.

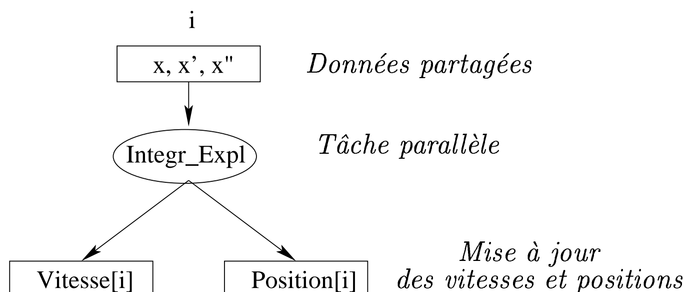


Figure 9.3 *Graphe de flots de données associé aux calculs des positions et vitesses relatives au bloc i via un schéma d'intégration explicite. Pour effectuer ces calculs, il suffit de connaître les états (position, vitesse et accélération) des particules de ce même bloc. Aucune information relative aux autres blocs n'est nécessaire.*

En résumé, cette tâche de calcul est similaire en terme de parallélisme à la tâche calculant les accélérations, car elle ne nécessite aucune information relative aux particules contenues dans les autres blocs. Cette tâche ne présente donc aucune dépendance en terme de données vis-à-vis des autres blocs.

Nous récapitulons pour conclure dans la figure 9.4 le graphe de flots de données construit au cours d'une itération de la simulation de textiles dans le cas de l'utilisation d'une intégration explicite.

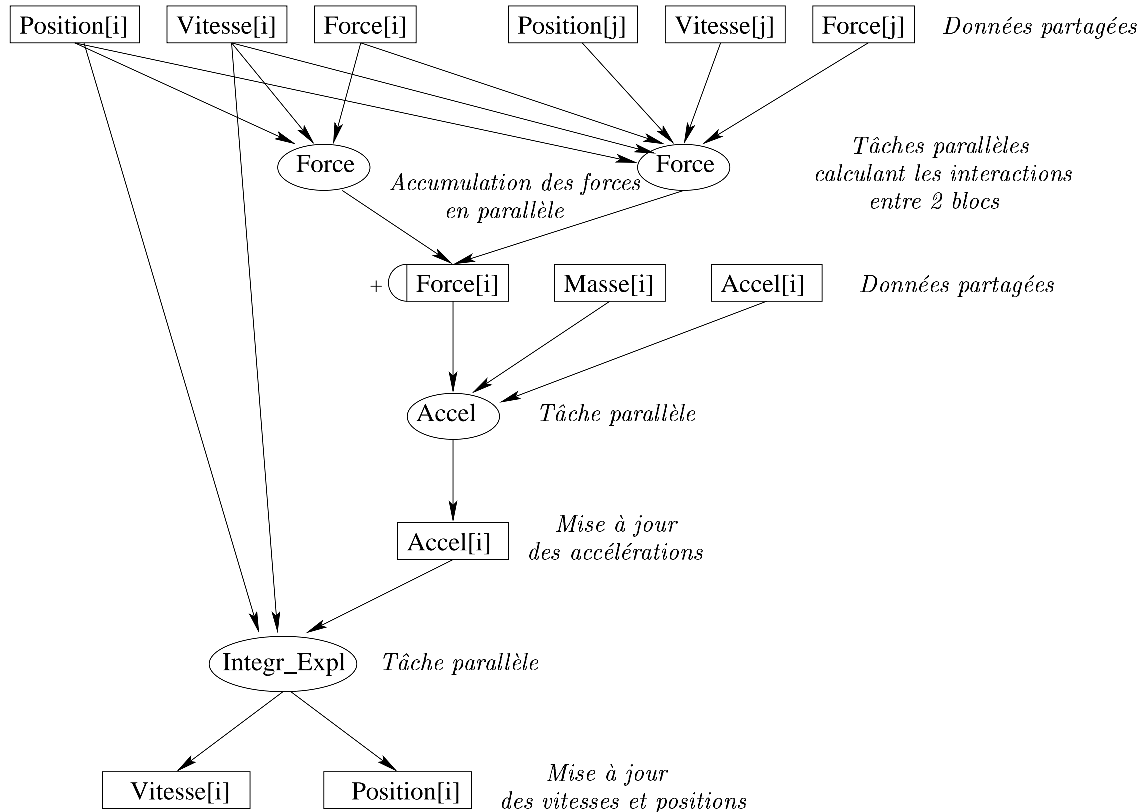


Figure 9.4 *Graphe de flots de données associé à une itération de la simulation de textiles dans le cas de l'utilisation d'une méthode d'intégration explicite*

En pratique, il est possible d'optimiser facilement le graphe de flots de données en regroupant la tâche d'accélération avec celle d'intégration : la tâche d'accélération est supprimée, le rapport des forces sur les masses étant réalisé dans la tâche d'intégration.

9.2.5 Schéma d'intégration implicite

Pour obtenir les vitesses et positions des particules, il est également possible d'intégrer les accélérations en utilisant un schéma implicite (cf. chapitre 3). Dans le cas de la simulation de textiles, il en résulte une meilleure stabilité. Par contre la charge de calcul engendrée est beaucoup plus importante que lors de l'emploi d'un schéma explicite avec notamment une résolution de système linéaire à effectuer. La méthode d'Euler implicite implantée peut se diviser en quatre étapes : (1) évaluer les matrices des dérivées des

forces, (2) construire un système d'équations linéaires creux, (3) résoudre ce système, et enfin (4) mettre à jour les vitesses et les positions des particules. Les algorithmes séquentiels relatifs à ces différentes étapes sont présentés dans la section 4.2.4 du chapitre 4.

Évaluation des matrices des dérivées des forces

La première étape dans l'application du schéma d'Euler implicite consiste à évaluer les matrices des dérivées des forces qui vont intervenir par la suite dans le système linéaire. Nous avons employé pour évaluer ces matrices les définitions de Volino et Magnenat-Thalmann (cf. chapitre 3, section 3.5.3).

Les matrices des contributions des forces $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial v}$ étant des matrices creuses, nous avons séparés dans deux structures différentes les éléments diagonaux des éléments non nuls se trouvant en dehors de la diagonale. La figure 9.5 illustre ces structures de données en reprenant la matrice des contributions présentées dans la section 4.2.4 par la figure 4.1 page 50. Les éléments diagonaux sont stockés dans un tableau fragmenté en blocs. Par exemple le bloc `Matrice_df_dx_diag[i]` représente le $i^{ème}$ bloc de la diagonale de la matrice $\frac{\partial f}{\partial x}$. Les éléments en dehors de la diagonale non nuls sont stockés dans une autre structure également fragmentée. Le $i^{ème}$ bloc de cette structure est noté `Matrice_df_dx[i][]`. Deux structures de données analogues ont été construites pour la matrice $\frac{\partial f}{\partial v}$. D'autre part, nous rappelons que chaque élément de ces matrices et donc chaque élément de ces structures de données, est une matrice de taille 3×3.

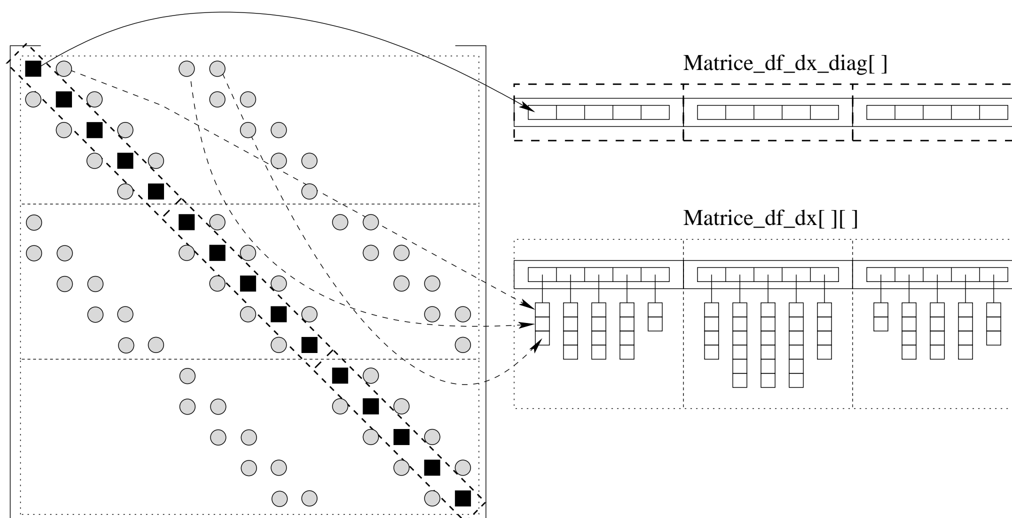


Figure 9.5 *Structures de données associées aux matrices des contributions des forces. Les éléments diagonaux sont séparés des éléments non diagonaux non nuls. Les deux structures employées sont fragmentées en blocs.*

Algorithme 15 Intégration en parallèle via le schéma d'Euler implicite (étape 1)

```
1 : struct Integr_Impl1 {
2 :   void operator() (a1::Shared_cw< Cumul, Vect6D > contrib1,
                      a1::Shared_cw< Cumul, Vect6D > contrib2,
                      a1::Shared_cw< Affect, VectMat6D > matContrib1,
                      a1::Shared_cw< Affect, VectMat6D > matContrib2,
                      a1::Shared_r< Vect3D> v1, a1::Shared_r< Vect3D > v2
                      a1::Shared_r< Vect3D > p1, a1::Shared_r< Vect3D > p2,
                      a1::Shared_r< VectInter> list_part) {
3 :     // Parcours de la liste des particules en interactions
4 :     for(int i=0; i<list_part.size(); ++i) {
5 :       // Contribution des forces entre list_part[i].part1
6 :       // et list_part[i].part2 : Contrib
7 :       tmpmatcontrib1[part1][num1] = Contrib;
8 :       tmpmatcontrib2[part2][num2] = Contrib;

9 :       // Accumulation des contributions pour chaque particule des 2 blocs
10 :       tmpvectcumul1[part1] += Contrib;
11 :       tmpvectcumul2[part2] += Contrib;
12 :     }

13 :     // Mise a jour des variables partagees
14 :     // Elements en dehors de la diagonale
15 :     matContrib1.cumul(tmpmatcontrib1);
16 :     matContrib2.cumul(tmpmatcontrib2);

17 :     // Elements de la diagonale
18 :     contrib1.cumul(tmpvectcumul1);
19 :     contrib1.cumul(tmpvectcumul2);
20 :   }
21 : };

22 : void Calcul_Integr_Impl1(Matrice_df_dx[][], Matrice_df_dx_diag[],
                            Position[], Vitesse[], list) {
24 :   // Liste des interactions entre les blocs : list
25 :   while (list.begin()!= list.end()) {
26 :     a1::Fork< Integr_Impl1 > () (list.contrib1, list.contrib2,
                                  list.matcontrib1, list.matcontrib2, list.v1,
                                  list.v2, list.p1, list.p2, list.list_part);
27 :     list++;
28 :   }
29 : }
```

Pour paralléliser le remplissage de ces structures avec les éléments des matrices, nous créons la tâche parallèle `Integr_Impl1` (algorithme 15, lignes 1-21), qui va évaluer les contributions des forces des particules de deux blocs de données. De la même manière que pour la tâche de calcul `Force` vue précédemment dans la section 9.2.2, cette tâche n'est instantiée que pour des paires de blocs ayant des interactions, c'est-à-dire pour des paires de blocs ayant des particules voisines (lignes 24-26). Pour cela nous utilisons une table de hachage similaire à celle présentée lors du calcul des forces (`list`, ligne 24). Les clés de la table correspondent aux paires de blocs adjacents et les données stockées pour une clé donnée sont la liste des particules interagissant entre ces deux blocs (`list_part`) et les blocs de données nécessaires aux calculs des contributions des forces (ligne 26). C'est-à-dire que pour la clé ($bloc_i$, $bloc_{j1}$) créée à cause des interactions existantes entre les sous ensembles i et j_1, les blocs i et j_1 des structures `Position[]`, `Vitesse[]`, `Matrice_df_dx_diag[]`, `Matrice_df_dv_diag[]`, `Matrice_df_dx[][]` et `Matrice_df_dv[][]` seront pointés dans la table de hachage.

Reprenons le même exemple que celui étudié pour le calcul des forces : nous supposons qu'il existe des interactions entre les particules du bloc i et celles du bloc j_1, et entre le bloc i et le bloc j_2. Notre but est de calculer les blocs i des éléments diagonaux et non diagonaux non nuls des matrices des contributions des forces. Observons l'algorithme 15 qui a été écrit pour la matrice $\frac{\partial f}{\partial x}$ mais dont les instructions seraient identiques pour $\frac{\partial f}{\partial v}$. Plusieurs tâches `Integr_Impl1` vont être créées à partir de la table de hachage pour traiter le bloc i (ligne 26). L'une comportera les données correspondant aux blocs i et j_1, une autre aura celles relatives aux blocs i et j_2 et une dernière aura les données relatives au seul bloc i, ce bloc possédant des particules en interactions. Toutes ces tâches seront ensuite exécutées en parallèle par ATHAPASCAN puisqu'elles sont précédées du terme `Fork`.

Au sein de chacune de ces tâches, les contributions des forces entre les particules des paires de blocs vont être calculées. Les éléments diagonaux des matrices représentent pour une particule donnée la somme de toutes les contributions des forces exercées par ses particules voisines. Tandis que les éléments non diagonaux représentent seulement la contribution des forces entre deux particules voisines.

A l'intérieur de la tâche `Integr_Impl1`, la liste des interactions entre les deux blocs considérées est parcourue (lignes 3-4). Pour chaque paire de particules `part1` et `part2` en interaction, le calcul des contributions des forces exercées l'une sur l'autre est effectué (`Contrib`, lignes 5-6). Il est dès lors possible de mettre à jour la valeur correspondante dans la matrice des contributions (lignes 7-8), et de l'additionner aux autres valeurs déjà obtenues pour ces particules dans le cas des éléments diagonaux (lignes 9-11). Pour cela des variables intermédiaires locales à la tâche sont employées, afin de mettre véritablement à jour les variables partagées en une seule fois à la fin de la tâche (lignes 13-19).

Le calcul de `Contrib` non détaillé dans l'algorithme est effectué à l'aide de la

connaissance des positions et vitesses des particules considérées à l'intérieur de la tâche. C'est pourquoi la tâche comporte en paramètres en lecture seule (`Shared_r`) les positions et vitesses des deux blocs pour lesquels la tâche est exécutée (ligne 2). Les variables partagées relatives aux blocs des éléments diagonaux et non diagonaux de la matrice sont eux pris en écriture cumulée (`Shared_cw`). Plusieurs processus peuvent ainsi modifier au même moment ces données partagées. Les éléments diagonaux sont réévalués à l'aide de la fonction `Cumul`, tandis que les éléments non diagonaux sont modifiés en employant la fonction `Affect` (ligne 2). La première additionne un à un les éléments du vecteur temporaire avec les éléments du bloc partagé, tandis que la seconde effectue une affectation, c'est-à-dire que les valeurs temporaires sont recopiées dans la variable partagée. En effet pour les éléments hors diagonale, plusieurs processeurs peuvent accéder en même temps à un bloc donné, mais aucun processeur ne modifie le même élément du bloc. Ces dépendances de données issues des tâches `Integr_Impl1` exécutées pour effectuer les calculs relatifs aux blocs i des matrices sont illustrées par la figure 9.6.

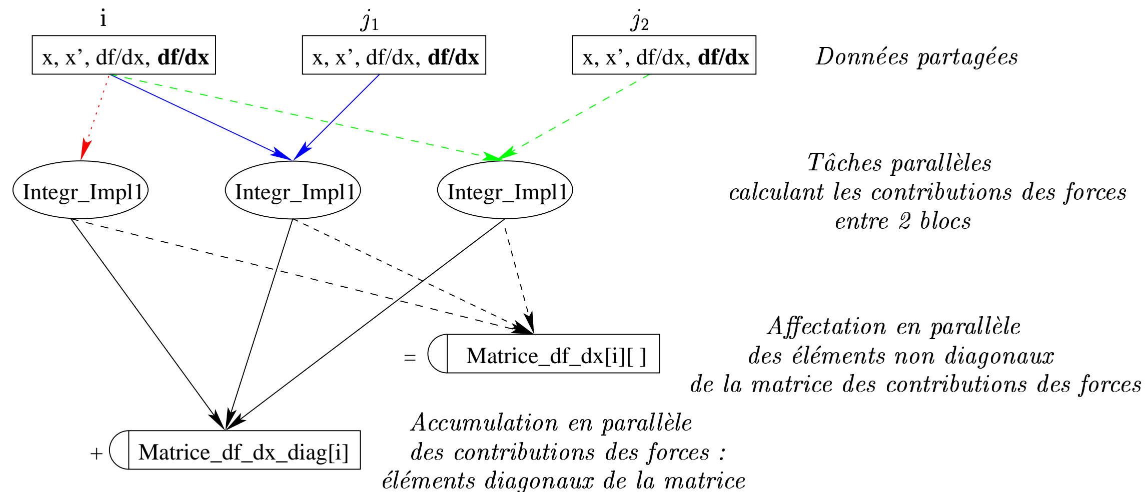


Figure 9.6 *Graphe de flots de données associé aux calculs des contributions des forces relatives au bloc i. Les particules contenues dans le $i^{ème}$ bloc ont des interactions avec celles disposées dans les blocs j_1 et j_2. Mais à ces interactions il ne faut pas oublier de rajouter les interactions locales au bloc i.*

En résumé le calcul relatif à un bloc des matrices des contributions des forces nécessite la connaissance des positions et vitesses des particules contenues dans les blocs voisins. Les tâches parallèles `Integr_Impl1` ont ainsi de fortes dépendances de données. Elles présentent donc les même dépendances que les tâches `Force` vues précédemment, calculant les interactions entre les particules contenues dans deux sous ensembles. D'un point de vue pratique, lors de l'emploi de la méthode d'intégration implicite, ces deux types de tâches ayant besoin des mêmes données, sont en fait combinées pour n'en faire plus

qu'une seule. Cette combinaison diminue ainsi le nombre de tâches parallèles en augmentant légèrement leur charge de calcul, et permet surtout de diminuer les communications au sein de la simulation parallèle.

Construction du système d'équations linéaires

A partir du moment où les blocs des matrices $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial v}$ sont calculés, il est possible de construire le système linéaire $Ax = b$. Il faut pour cela remplir la matrice $A = M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}$ et le vecteur $b = hf(t) + h^2\frac{\partial f}{\partial x}v(t)$. L'obtention de ce système linéaire à partir du schéma d'Euler implicite est expliquée au sein de la section 3.5.3, page 41.

Le vecteur b est stocké dans la structure de données `B[ ]`. Cette structure est fragmentée de la même façon que les autres, en blocs de données partagées. Le remplissage du vecteur `B[ ]` s'élabore en deux étapes. Il faut tout d'abord effectuer la multiplication entre la matrice $\frac{\partial f}{\partial x}$ et le vecteur $v(t)$, avant de remplir entièrement le vecteur b en appliquant sa définition. La multiplication entre la matrice $\frac{\partial f}{\partial x}$ et le vecteur $v(t)$ s'effectue en parallèle à l'aide de deux types de tâches. Les premières effectuent la multiplication du vecteur $v(t)$ avec les éléments diagonaux de la matrice, tandis que les secondes effectuent la même multiplication mais avec les éléments non diagonaux. Les résultats sont cumulés au fur et à mesure afin d'obtenir au final dans le vecteur `B[ ]` le résultat de la multiplication de $\frac{\partial f}{\partial x}$ par $v(t)$.

La tâche `Integr_Impl2_Diag` effectue la multiplication entre un bloc contenant les éléments diagonaux de la matrice `Matrice_df_dx_diag[]` et le bloc correspondant du vecteur des vitesses (algorithme 16 , lignes 1-8). Cette tâche est similaire à celle des accélérations dans le sens où le calcul relatif au bloc i de `B[]` ne nécessite aucune information sur les autres blocs (ligne 11) et qu'il y aura autant de tâches instantiées que de blocs présents dans la simulation (lignes 10-11).

Les blocs `Vitesse[]` et `Matrice_df_dx_diag[]` au sein de cette tâche sont pris en lecture seule (`Shared_r`, ligne 2), tandis que le bloc du vecteur `B[]` est pris en écriture cumulée (`Shared_cw`, ligne 2), afin de laisser la possibilité aux processeurs exécutant le second type de tâche relatif à ce calcul d'y additionner les multiplications effectuées avec les éléments non diagonaux.

Pour chaque élément (de taille 3) d'un bloc `B[]` donné, le produit entre l'élément correspondant du vecteur vitesse (de taille 3) et celui de la matrice (de taille 3×3) est effectué (lignes 3-4). Les résultats de ces calculs sont tout d'abord stockés dans un vecteur temporaire (ligne 4), pour ensuite mettre à jour la variable partagée en une fois, dès que tous les éléments du bloc ont été calculés (lignes 5-6) en appliquant la fonction `Cumul` (ligne 2).

Algorithme 16 Intégration en parallèle via le schéma d'Euler implicite (étape 2) : remplissage du vecteur `B[ ]` à partir des éléments diagonaux de la matrice $\frac{\partial f}{\partial x}$

```
1 : struct Integr_Impl2_Diag {
2 :   void operator() (a1::Shared_r< Vect6D > contrib, a1::Shared_r< Vect3D > v,
                    a1::Shared_cw< Cumul, Vect3D > b) {
3 :        for(int i=0 ; i<v.read().size() ; ++i)
4 :            tmpb[i] = contrib.read()[i] * v.read()[i] ;

5 :        // Mise a jour de la variable partagee
6 :        b.cumul(tmpb) ;
7 :     }
8 : } ;

9 : void Calcul_Integr_Impl2_Diag(Matrice_df_dx_diag[], Vitesse[],
                                  B[], Nb_Blocs) {
10 :   for(int i=0 ; i<Nb_Blocs ; ++i)
11 :     a1::Fork< Integr_Impl2_Diag > () (Matrice_df_dx_diag[i],
                                          Vitesse[i], B[i]) ;
12 : }
```

Pour obtenir le résultat final de la multiplication de la matrice $\frac{\partial f}{\partial x}$ par le vecteur $v(t)$, nous rajoutons le résultat de la multiplication du vecteur des vitesses par les éléments non diagonaux de la matrice. Pour cela la tâche `Integr_Impl2_NonDiag` est seulement instantiée pour des paires de blocs ayant des interactions (algorithme 17, lignes 14-16).

En effet, lors du produit matrice/vecteur, les éléments de $v(t)$ seront seulement multipliés par les éléments non diagonaux de la matrice avec lesquels ils ont un lien (figure 9.7). Cette tâche effectue ainsi la multiplication entre les blocs contenant les éléments non diagonaux de la matrice et les blocs correspondant du vecteur des vitesses (lignes 1-8). Les résultats étant cumulés au fur et à mesure dans les blocs correspondant du vecteur `B[ ]` (lignes 8-10) grâce à un accès des variables partagées par écriture cumulée (ligne 2).

Cette tâche suivant le même principe que celle calculant les forces (section 9.2.2) et celle évaluant les matrices des contributions des forces (section 9.2.5), nous ne la détaillerons pas plus.

Les figures 9.7 et 9.8 présentent les tâches de calculs parallèles relatives aux calcul du vecteur `B[i]`. Ces schémas reprennent l'exemple présenté pour les forces et repris pour l'évaluation des matrices des contributions. C'est-à-dire que le bloc i, objet du calcul, possède des interactions avec les blocs i, j_1 et j_2. La figure 9.7 illustre plus particulièrement l'obtention du bloc `B[i]` en reprenant la configuration présentée précédemment de la matrice des contributions des forces. Tandis que la figure 9.8 présente le graphe de flots de données de l'ensemble des tâches nécessaires au calcul du bloc `B[i]`.

Algorithme 17 Intégration en parallèle via le schéma d'Euler implicite (étape 2) : remplissage du vecteur b à partir des éléments non diagonaux de la matrice

```
1 : struct Integr_Impl2_NonDiag {
2 :   void operator() ( a1::Shared_r< Vect3D > v1, a1::Shared_r< Vect3D > v2,
                      a1::Shared_cw< Cumul, Vect3D > b1,
                      a1::Shared_cw< Cumul, Vect3D > b2,
                      a1::Shared_r< Vect6D > matcontrib1,
                      a1::Shared_r< Vect6D > matcontrib2, list_part) {
3 :     // Parcours de la liste des particules en interactions
4 :     for(int i=0 ; i<list_part.size() ; ++i) {
5 :       tmpb1[part1] += matcontrib1[part1][num1].read()[i] * v1.read()[part2] ;
6 :       tmpb2[part2] += matcontrib2[part1][num2].read()[i] * v2.read()[part1] ;
7 :     }

8 :     // Mise a jour des variables partagees
9 :     b1.cumul(tmpb1) ;
10 :     b2.cumul(tmpb2) ;
11 :   }
12 : } ;

13 : void Calcul_Integr_Impl2_NonDiag(Matrice_df_dx[][], Vitesse[], B[], list) {
14 :   // Liste des interactions entre les blocs : list
15 :   while (list.begin() != list.end()) {
16 :     a1::Fork< Integr_Impl2_Diag > () (list.v1, list.v2, list.b1, list.b2
                           list.matcontrib1, list.matcontrib2, list.list_part) ;
17 :     list++ ;
18 :   }
19 : }
```

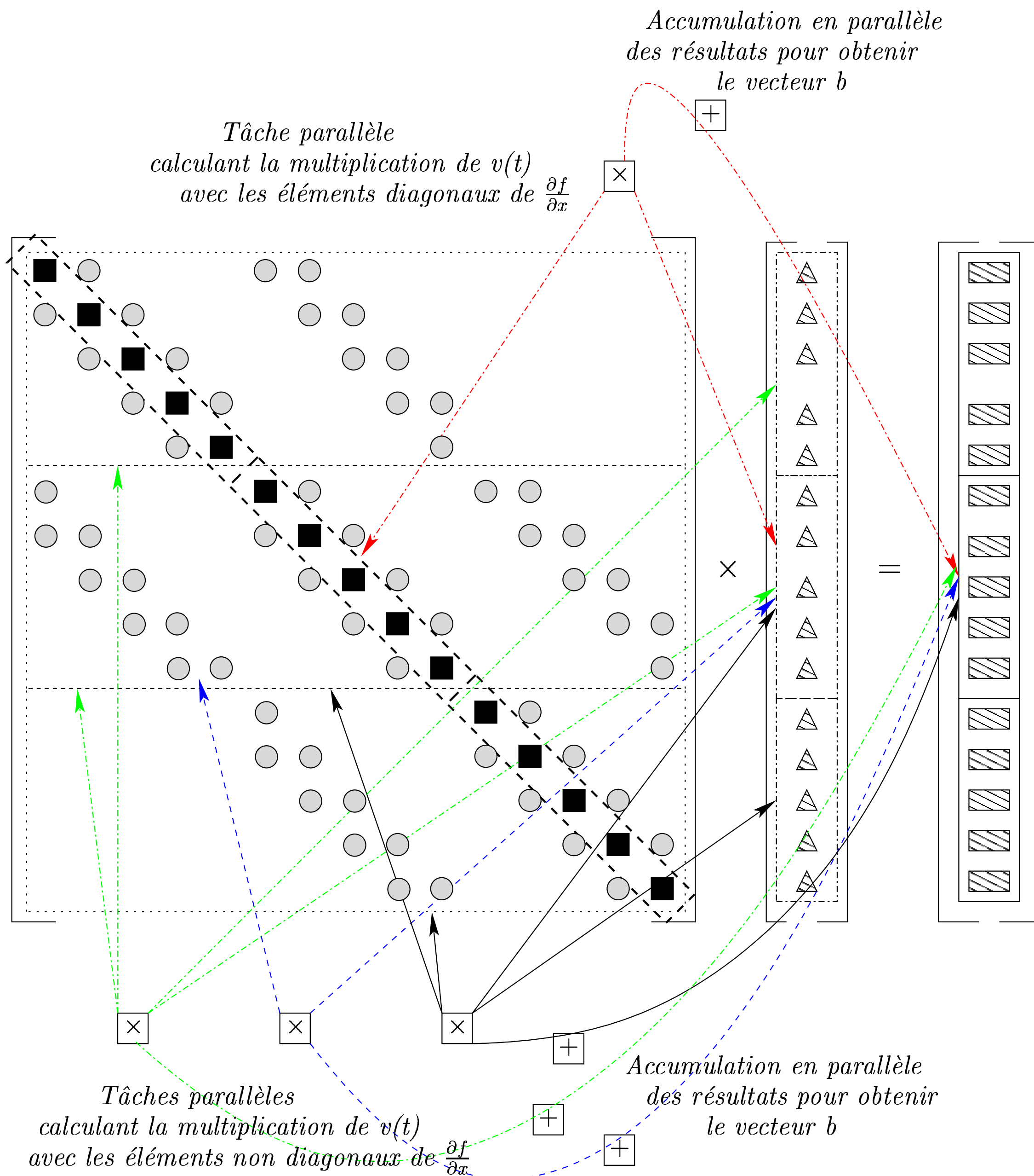


Figure 9.7 *Première étape du calcul en parallèle du vecteur b. Deux types de tâches sont créées. Les premières effectuent la multiplication des éléments diagonaux de $\frac{\partial f}{\partial x}$ par le vecteur $v(t)$. Les secondes effectuent la même multiplication mais avec les éléments non diagonaux de la matrice. Les résultats sont additionnés au fur et à mesure dans le vecteur b.*

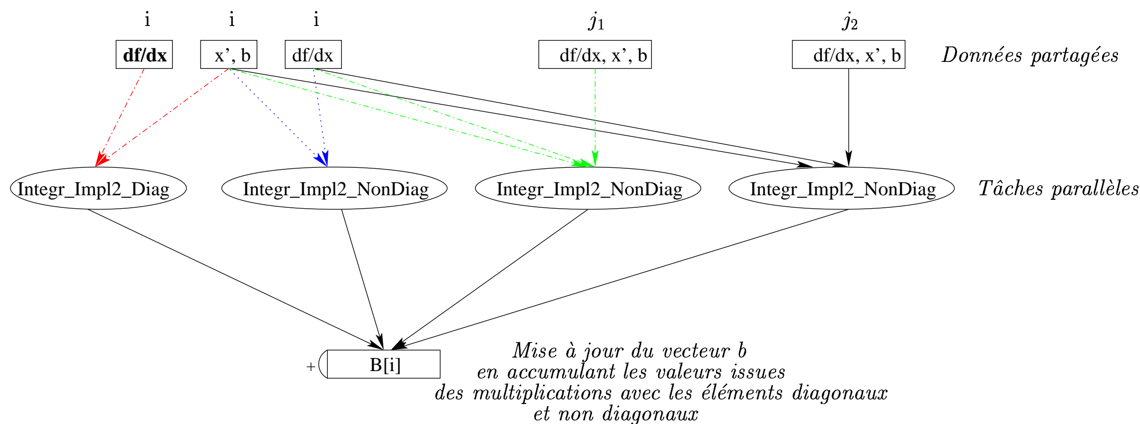


Figure 9.8 *Graphe de flots de données associé au calcul du bloc i du vecteur b, multiplication des éléments de la matrice des contributions avec le vecteur vitesse. Deux types de tâches permettent l'obtention du résultat de ce calcul. Les premières multiplient les éléments diagonaux de la matrice au vecteur $v(t)$, tandis que les secondes traitent les éléments non diagonaux de la matrice.*

Une fois que la multiplication de la matrice $\frac{\partial f}{\partial x}$ par le vecteur $v(t)$ est finie, la tâche parallèle `Integr_Impl2` réalise la dernière phase du remplissage du vecteur `B[]` en appliquant tout simplement sa définition $b = hf(t) + h^2\frac{\partial f}{\partial x}v(t)$ (algorithme 18, lignes 1-6). Cette tâche est instantiée pour chacun des blocs `B[]` contenus dans le système (lignes 8-9).

Algorithme 18 Intégration en parallèle via le schéma d'Euler implicite (étape 2) : phase finale du remplissage du vecteur `B[]`

```
1 : struct Integr_Impl2 {
2 :   void operator() (a1::Shared_r< Vect3D > m, a1::Shared_r< Vect3D > f,
                      a1::Shared_r_w< Vect3D > b, double h) {
3 :          for(int i=0; i<m.read().size(); ++i)
4 :              b.access()[i] = (h * f.read()[i]) + (h * h * b.access()[i]);
5 :       }
6 : };

7 : void Calcul_Integr_Impl2(Masse[], Force[], B[], h, Nb_Blocs) {
8 :   for(int i=0; i<Nb_Blocs; ++i)
9 :     a1::Fork< Integr_Impl2 > () (Masse[i], Force[i], B[i], h);
10 : }
```

Le graphe de flots de données associé au traitement par cette tâche du bloc `B[i]` est présenté par la figure 9.9. Cette tâche ne présente aucune dépendance de données vis-à-

vis des autres blocs. Il suffit de connaître les masses et les forces des particules contenues dans le bloc i considéré, ainsi que les valeurs du bloc `B[i]` calculées précédemment (ligne 2) afin de terminer le remplissage de ce dernier (lignes 3-4).

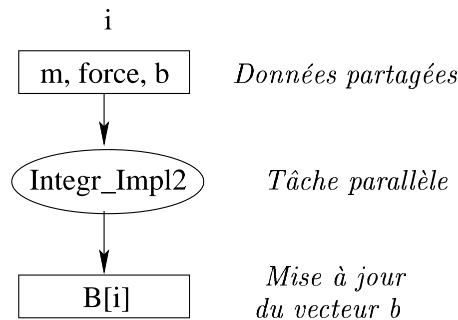


Figure 9.9 *Graphe de flots de données associé à la dernière tâche de calculs des éléments du bloc `B[i]`. Aucune information relative aux autres blocs n'est nécessaire.*

Afin de terminer la conception du système linéaire, il reste normalement à remplir une matrice A en appliquant la définition $A = M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}$. Mais en pratique nous nous servons directement des matrices des contributions des forces sans passer par une structure intermédiaire inutile et coûteuse en terme de place mémoire.

En résumé, la construction du système linéaire requiert essentiellement le remplissage du vecteur b au sein de la structure `B[]` décomposée en blocs de données partagées. Le remplissage de ce vecteur passe par le calcul de la multiplication de la matrice des contributions des forces $\frac{\partial f}{\partial x}$ par le vecteur $v(t)$. Pour cela deux types de tâches de calculs parallèles sont à exécuter. Les premières traitant un bloc spécifique d'éléments diagonaux de la matrice, sont totalement indépendantes vis-à-vis des informations contenues dans les autres blocs. Les secondes s'occupant des éléments non diagonaux de la matrice traitent des paires de blocs adjacents, faisant apparaître des dépendances de données entre ces blocs. Mais au sein de ces deux types de tâches, les blocs de la structure `B[]` étant pris en écriture cumulée, toutes ses tâches peuvent être exécutées au même moment par des processeurs différents, mettant chacun à jour les valeurs du vecteur `B[]`. Le remplissage de ce vecteur se termine par l'élaboration d'un dernier type de tâche, appliquant la définition du vecteur à chacun des éléments contenus dans un bloc, ne nécessitant que des informations relatives à ce dernier.

Résolution par la méthode du Gradient Conjugué

L'emploi du schéma d'Euler implicite conduit à la résolution du système linéaire étudié précédemment. La méthode du Gradient Conjugué tirant partie de la non densité des

matrices est particulièrement adaptée dans notre cas. L'algorithme séquentiel de cette méthode a été présenté brièvement dans la section 4.2.4, et de manière plus détaillée dans le chapitre A des annexes. Cet algorithme, décrit plus simplement par l'algorithme 19, ne présente que des calculs d'algèbres linéaires tels que des produits scalaires, des produits matrice $\times$ vecteur et des multiplications de vecteurs par des scalaires.

Algorithme 19 Algorithme simplifié du Gradient Conjuguée

```
1 :   for(int i=0; i<nb_iter; ++i) {
2 :      CalculNorme();
3 :      CalculDirection();
4 :      CalculProdMatVect();
5 :      CalculAlpha();
6 :      CalculSolution_Residu();
7 :   }
```

La fonction `CalculNorme` (ligne 2) effectue le calcul de la norme d'un vecteur, soit le produit entre la transposée du vecteur et lui-même. Il en est de même pour la fonction `CalculAlpha` (ligne 5). Pour paralléliser ces calculs, il suffit de créer deux tâches instantiées autant de fois qu'il y a de blocs composant le vecteur global. Pour un bloc donné, ces tâches calculent pour chacun des éléments du bloc le produit de cet élément avec lui-même. Le résultat de chacune des multiplications est cumulé au fur et à mesure dans une variable partagée accédée en écriture cumulée. Ces tâches ne nécessitent ainsi aucune autre information extérieure au bloc traité.

La fonction `CalculDirection` (ligne 3) met à jour le vecteur des directions. Ce calcul consiste à effectuer la multiplication d'un vecteur par un scalaire et de l'additionner à un autre vecteur. Il en est de même pour la fonction `CalculSolution_Residu` (ligne 6) qui met à jour le vecteur de solution et celui des résidus. Ces opérations peuvent être parallélisées en créant des tâches, instantiées autant de fois qu'il y a de blocs constituant les vecteurs. Puis au sein de ces tâches l'opération souhaitée est effectuée pour chacun des éléments du bloc. Ces tâches ne présentent ainsi aucune dépendances de données. D'autre part les variables partagées correspondant au bloc sur lesquels les tâches opèrent sont pris en simple lecture/écriture, les autres processeurs ne nécessitant pas d'accès à ces variables tant qu'elles ne sont pas totalement mises à jour.

La fonction `CalculProdMatVect` (ligne 4) effectue la multiplication de la matrice A par le vecteur des directions, soit $\left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right) \times d$. Pour cela, de la même manière que lors du calcul du produit de $\frac{\partial f}{\partial x}$ par le vecteur $v(t)$, trois tâches de calcul sont créées. La première, sur le même principe que la tâche `Integr_Impl2_Diag` vue précédemment, effectue à la fois le produit entre les éléments diagonaux de $\frac{\partial f}{\partial x}$ et les éléments diagonaux de $\frac{\partial f}{\partial v}$ par le vecteur d. Cette tâche est instantiée autant de fois qu'il y a de blocs dans le système et ne présente pas de dépendances de données vis à vis des autres blocs. La seconde suit la même démarche que la tâche `Integr_Impl2_NonDiag` afin

d'effectuer le produit des éléments non diagonaux des matrices $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial x}$ par le vecteur de direction. C'est-à-dire que cette tâche est instantiée pour chaque paire de blocs ayant des interactions, et que les données partagées accédées en écriture/cumulée sont mises à jour au même instant par différents processeurs cumulant leurs résultats. Cette tâche nécessite donc des informations relatives à deux blocs adjacents. Enfin la troisième, suivant la même idée que la tâche `Integr_Impl2`, termine le calcul du produit $A \times d$ en appliquant pour chacun des éléments d'un bloc donné la définition de ce produit à partir des parties précédemment calculées. Cette tâche ne nécessite ainsi que des informations relatives à un bloc donné.

En résumé, la parallélisation de l'algorithme de la méthode du Gradient Conjugué présente les deux types de tâches déjà vues au préalable. A savoir des tâches instantiées autant de fois que le nombre de blocs, traitant un à un les éléments de ces blocs sans aucune information sur les autres blocs, et des tâches qui par contre sont instantiées pour deux blocs ayant des interactions pris en écriture cumulée afin que plusieurs processeurs puissent les modifier au même moment grâce à des fonctions commutatives des paramètres et associatives.

Mise à jour des positions et vitesses

La résolution du système linéaire a permis l'obtention des valeurs du vecteur Δv. Ces données sont stockées dans le tableau `X[]`. Il est dès lors possible de mettre à jour les positions et vitesses de l'ensemble des particules.

Pour effectuer la parallélisation de ce calcul, la tâche parallèle `Integr_Impl4` est créée (algorithme 20, lignes 1-8). Cette tâche va mettre à jour les vitesses et positions de l'ensemble des particules contenues dans un bloc. Cette mise à jour est faite à partir du bloc du vecteur solution correspondant obtenu après la résolution du système linéaire. De la même façon que pour les tâches calculant les accélérations et celles effectuant une intégration explicite, il y a autant de tâches parallèles `Integr_Impl4` instantiées que de blocs (lignes 10-11). En effet, cette tâche de calcul est identique d'un point de vue algorithmique parallèle que celles vues précédemment pour la méthode d'intégration explicite et pour le calcul des accélérations.

Considérons le cas de la mise à jour des blocs i. Une tâche parallèle `Integr_Impl4` est instantiée ayant en paramètres en lecture seule (`Shared_r`) le bloc contenant les solutions Δv des particules du bloc i, et en lecture/écriture (`Shared_r_w`) les blocs relatifs aux positions et vitesses de ces particules (ligne 2). La tâche effectue ensuite pour chacune des particules du sous ensemble (ligne 3), le calcul de leurs vitesses (ligne 5) à partir du bloc du vecteur solution correspondant, ainsi que le calcul de leurs positions (ligne 4) en employant le schéma d'Euler explicite nécessitant le pas de temps h.

Algorithme 20 Intégration en parallèle via le schéma d'Euler implicite (étape 4)

```
1 :   struct Integr_Impl4 {
2 :     void operator() (a1::Shared_r_w< Vect3D > p, a1::Shared_r_w< Vect3D > v,
                         a1::Shared_r< Vect3D > x, double h) {
3 :            for(int i=0 ; i<p.access().size() ; ++i) {
4 :               v.access()[i] = v.access()[i] + x.read()[i] ;
5 :               p.access()[i] = p.access()[i] + h * v.access()[i] ;
6 :            }
7 :         }
8 :   } ;
9 :   void Integration_Impl4(Position[], Vitesse[], X[], h, Nb_Blocs) {
10 :      for(int i=0 ; i<Nb_Blocs ; ++i)
11 :          a1::Fork< Integr_Impl4 > () (Position[i], Vitesse[i], X[i], h) ;
12 :  }
```

Le bloc relatif aux solutions étant pris en lecture, ses données sont accédées via la méthode `read()` (ligne 4). La méthode `access()` est par contre employée pour les blocs des positions et vitesses dont les données, prises en lecture/écriture, doivent être modifiées (lignes 4-5). Le graphe de flots de données associé à ces calculs est représenté par la figure 9.10.

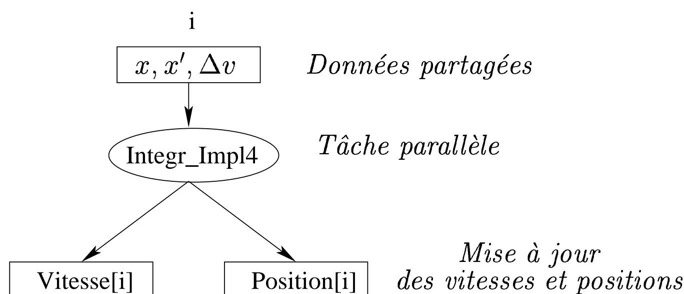


Figure 9.10 *Graphe de flots de données associé à la mise à jour des positions et vitesses relatives au bloc i via un schéma d'intégration implicite. Pour effectuer ces calculs, il suffit de connaître les positions et vitesses des particules de ce même bloc, ainsi que le bloc solution Δv correspondant. Aucune information relative aux autres blocs n'est nécessaire.*

En conclusion, cette tâche de calcul est similaire en terme de parallélisme à la tâche calculant les accélérations et à celle calculant les positions et vitesses via un schéma explicite. En effet, cette tâche ne nécessite aucune information relative aux particules contenues dans les autres blocs. Elle ne présente donc aucune dépendance en terme de données vis-à-vis des autres blocs.

9.3 Ordonnancement

Dans un programme ATHAPASCAN, le parallélisme est exprimé par la création de tâches asynchrones, l'ordre de référence lexicographique définissant les synchronisations entre ces tâches. Toute exécution doit respecter cette sémantique définissant un ordre partiel sur l'exécution des tâches et les accès aux données partagées.

En effet, durant l'exécution, le graphe de flots de données associé au programme est construit par ATHAPASCAN. Ce graphe est constitué des tâches et des données partagées définies dans le programme. Ensuite, à partir de ce graphe, les algorithmes d'ordonnancement ont la responsabilité de placer les données partagées et de distribuer l'ensemble des tâches créés parmi les processeurs disponibles. Ceci est effectué avec un souci de performance aussi bien au niveau de la mémoire utilisée que du temps d'exécution. La phase d'ordonnancement peut ainsi être vue comme une phase d'optimisation de l'exécution.

9.3.1 Contrôle de l'ordonnanceur

Dans la majorité des programmes ATHAPASCAN, le graphe de flots de données est construit au début de l'exécution du programme pour que ensuite l'ordonnanceur se charge de placer les tâches et les données partagées sur les processeurs qui vont par la suite être exécutées dans le respect des contraintes de précédences de données. Pour des programmes de type simulation possédant une *boucle infinie*, il est impossible d'attendre la fin de la construction du graphe de taille infinie pour débuter l'exécution des tâches.

Algorithme 21 Contrôle de l'ordonnanceur d'ATHAPASCAN

```
1 :   // Boucle infinie de la simulation de textiles
2 :   for (int j=0 ; j< 100 000 ; j++){
3 :      // Creation des taches et construction du graphe de flots de donnees
4 :      for (int i=0 ; i<Nb_Blocs ; i++)
5 :            a1::Fork< Accel > () (Force[i], Masse[i], Accel[i]) ;
6 :      for (int i=0 ; i<Nb_Blocs ; i++)
7 :            a1::Fork< Integr_Expl > () (Position[i], Vitesse[i], Accel[i], h) ;
8 :      ...

9 :      // Execution des taches creees toutes les N iterations
10 :      if (j % N)
11 :        com->sync() ;
12 :   }
```

ATHAPASCAN fournit alors une “macro” permettant à l'utilisateur de gérer la *fréquence d'ordonnancement du graphe* ainsi que son interruption temporaire afin de rendre possible l'exécution des tâches déjà créées dans le graphe partiellement construit. Cette

fréquence de l'ordonnancement doit être suffisamment grande pour éviter la construction d'un graphe trop important en terme de place mémoire, mais également suffisamment petite pour assurer un nombre minimal de tâches créées permettant une prédiction correcte de l'exécution future du programme considéré.

Ce contrôle de l'ordonnanceur est effectué via une annotation dans le code du programme ATHAPASCAN indiquant pendant l'exécution le fait qu'un nombre suffisant de tâches ont été incorporées dans le graphe. L'algorithme 21 présente cette annotation. Le programme de la simulation de textiles est composé d'une boucle infinie (ligne 2) à l'intérieur de laquelle des tâches sont créées dynamiquement, ayant en paramètres les données partagées définies dans la simulation (lignes 3-8). Le graphe de flots de données correspondant à la simulation est alors construit au fur et à mesure par l'environnement ATHAPASCAN en incorporant une à une les tâches créées.

Cette construction s'arrête à la ligne 11 avec l'instruction `sync()` qui force l'ordonnancement des tâches déjà créées et qui impose l'attente de leur fin d'exécution avant de recommencer à créer de nouvelles tâches et à construire un nouveau graphe. La génération des communications est également effectuée au moment de cette instruction. Dans cet exemple, chaque graphe de flots de données partiel correspond aux tâches crées durant `N` itérations de la simulation.

9.3.2 Annotations de code

Le développeur d'une application basée sur l'environnement ATHAPASCAN possède le contrôle total du choix des algorithmes d'ordonnancement employés durant l'exécution de son application. La bibliothèque ATHAPASCAN fournit un certain nombre d'algorithmes d'ordonnancement, mais il est également possible d'intégrer son propre algorithme d'ordonnancement.

Chaque tâche de l'application possède un ordonnancement par défaut (algorithme 22, lignes 1-2). Cet ordonnancement est attribué lors de la création de la tâche via un paramètre de l'opérateur `Fork`. Cette association est effectuée implicitement : une tâche créée à partir d'une autre tâche possède le même ordonnancement que sa tâche mère.

Algorithme 22 Annotation de l'opérateur `Fork` avec des paramètres d'ordonnancement

```
1 : // Creation de la tache ATHAPASCAN Accel avec l ordonnancement par defaut
2 : a1::Fork< Accel > () (Force[i], Masse[i], Accel[i]);

3 : // Creation de la tache ATHAPASCAN Accel
4 : // en specifiant des parametres pour l ordonnancement
5 : a1::Fork< Accel > ( [sched_attributes] ) (Force[i], Masse[i], Accel[i]);
```

Certains algorithmes d'ordonnancement requièrent des informations supplémentaires sur les tâches (coût, localité, priorité). Ces informations sont alors fournies lors de la création de la tâche en annotant l'opérateur `Fork` avec des paramètres d'ordonnancement

en utilisant le champs `sched_attributes` (algorithme 22, lignes 3-5). Ces attributs dépendant de l'ordonnancement sont rajoutés à titre optionnel et ne modifient en rien le résultat de l'exécution.

ATHAPASCAN fournit comme politique d'ordonnancement les deux paramètres suivants : *OwnerComputeRule* et *HPF*. *OwnerComputeRule* permet de spécifier à l'ordonnanceur que la tâche doit être exécutée en priorité sur le noeud possédant une certaine donnée partagée. Dans l'algorithme 23, lignes 7-9, le paramètre `OCR(Accel[i])` spécifie que la tâche `Accel` doit être en priorité exécutée sur le noeud où se trouve la donnée partagée `Accel[i]`, ceci afin d'éviter des communications inutiles de données.

Algorithme 23 Spécification des politiques d'ordonnancement

```
1 : // Choix de la distribution des blocs de particules sur p noeuds
2 : a1::BlocCyclic1D BC(p) ;

3 : // Placement de la tache d'initialisation Init
4 : // selon le choix specifie par a1::Distribute(BC,i) :
5 : // la tache associee au bloc i est placee selon la politique de BC
6 : a1::Fork< Init > ( a1::Distribute(BC,i) ) (Accel[i]) ;

7 : // Placement de la tache Accel selon la politique OwnerComputeRule :
8 : // la tache est executee sur le noeud possedant la donnee partagee Accel[i]
9 : a1::Fork< Accel > ( OCR(Accel[i]) ) (Force[i], Masse[i], Accel[i]) ;
```

HPF permet quant à lui de spécifier la distribution des tâches en utilisant une fonction de placement par rapport à l'indice des noeuds. Dans l'algorithme 23, lignes 1-2 et 3-6, une distribution cyclique des blocs des données partagées contenues dans le vecteur `Accel[]` est effectuée à l'initialisation sur une grille $p \times 1$ processeurs. Les lignes 1-2 spécifient la façon dont la distribution est effectuée sur la grille et les lignes 3-6 donnent ces informations à l'ordonnanceur en spécifiant lors de la création de la tâche `Init` l'indice du bloc considéré. Enfin, la tâche `Accel` est créée selon la politique OwnerComputeRule appliquée à la donnée partagée `Accel[i]`, afin qu'elle soit exécutée sur le processeur possédant cette donnée.

L'ordonnancement employé pour une application est spécifié dans la commande d'exécution du programme. A l'heure actuelle, trois algorithmes d'ordonnancement sont disponibles dans ATHAPASCAN :

- *Cyclic*, qui assure une stratégie d'allocation de type tourniquet (*round robin*) des tâches créées.
- *Largest Processing Time First (LPTF)*, qui permet d'allouer les tâches cycliquement sur les processeurs, après les avoir trié en fonction de leur coût (nombres d'instruc-

tions contenues dans la tâche). Pour pouvoir utiliser cet ordonnancement, les tâches doivent être au préalable annotées avec une évaluation de leur temps d'exécution.
– *Orthogonal Recursive Bissection (ORB)*, effectuant un découpage orthogonal récursif de l'espace géométrique défini par la position des particules dans notre cas. Cet algorithme nécessite des informations sur les coûts des tâches et sur la localisation des données dans l'espace considéré.

9.3.3 Algorithmes d'ordonnancement de graphes

Dans le but d'obtenir un placement efficace des données sur les processeurs, il est possible d'utiliser des algorithmes d'ordonnancement de graphes permettant le partitionnement du graphe de dépendances. Ce placement s'effectue alors en trois étapes :

1. Le graphe de flots de données est tout d'abord converti en graphe de dépendances (étape de la descente).
2. Son partitionnement est ensuite effectué en utilisant la bibliothèque Scotch [94, 93] (étape de partitionnement).
3. Enfin, une heuristique est employée pour calculer un placement des tâches contenues dans le graphe de flots de données (étape de la remontée).

L'étape 1 est rendue possible par le fait que le graphe de flots de données apporte des informations plus détaillées que le graphe de dépendances. Chaque donnée partagée dans le graphe de flots de données représente un noeud dans le graphe de dépendances. Puis pour chaque couple de données partagées accédé par une tâche dans le graphe de flots de données, un arcs est inséré entre les noeuds du graphe de dépendances. Par ailleurs, les arcs du graphe de dépendances sont annotés de coûts établis à partir d'informations données au moment du `Fork` de la tâche. Les coûts des noeuds sont eux calculés en utilisant les volumes de communications (la taille des données partagées).

Après l'appel des fonctions de la bibliothèque Scotch, une partition des noeuds est calculée. Il s'agit de l'attribution d'un processeur à chaque noeud du graphe de dépendances. Le placement retourné par Scotch correspond donc à celui des données partagées. A partir de là, le but de l'étape 3 est de placer les tâches sur les processeurs : étant donnés les sites de placement des données partagées accédées par une tâche, sur quel site placer cette tâche ? Si la tâche ne possède qu'un seul accès sur une donnée partagée, il est intéressant de la placer sur le même processeur que la donnée : la règle 'OwnerComputeRule" est alors logiquement employée. Par ailleurs, si la tâche possède plusieurs accès à des données partagées placées sur différents processeurs, il est encore possible d'employer la règle "OwnerComputeRule" : la tâche est alors placée sur le processeur possédant le plus grand nombre de ses paramètres. L'heuristique employée vise alors à augmenter la localité des données. D'autres heuristiques peuvent être employées pour cette étape de

placement des données. Certaines peuvent avoir pour objectif la minimisation des communications en prenant en compte par rapport à l'heuristique précédente, le coût estimé du déplacement de données partagées vers le site associé à la tâche considérée. Le but peut encore être d'équilibrer la charge de calcul sur l'ensemble des processeurs. Pour cela la tâche considérée est affectée au processeur le moins chargé. Enfin une autre heuristique peut être imaginée tirant partie des deux précédentes : la tâche serait alors placée en essayant de trouver le meilleur compromis entre l'équilibrage de la charge des processeurs et la minimisation des communications engendrées par son placement.

La figure 9.11 présente le partitionnement du graphe de flots de données de la simulation de textiles (figure 7.2) obtenu en employant la bibliothèque Scotch. Ce partitionnement a été réalisé pour placer 16 tâches de calcul de la simulation sur 4 processeurs. Les cercles représentent les tâches de calculs, les rectangles les processeurs, et les traits les dépendances entre les tâches.

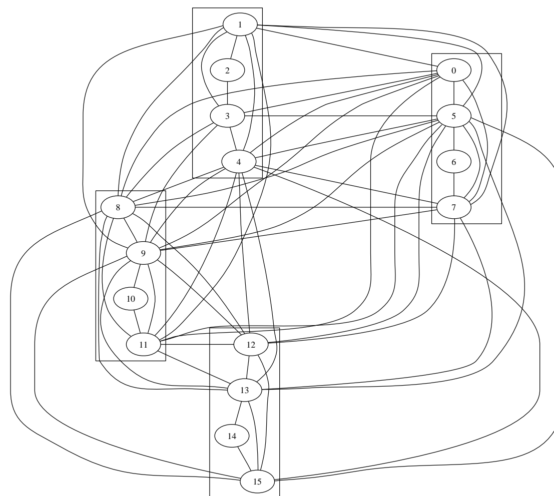


Figure 9.11 *Partitionnement du graphe de dépendances de la simulation de textiles*

9.4 Conclusion

Le parallélisme au sein de l'environnement de programmation parallèle ATHAPASCAN est décrit via la création de tâches de calculs asynchrones. Cette description est totalement indépendante de la machine cible sur laquelle les tâches seront par la suite exécutées, assurant une totale portabilité des programmes conçus. Les contraintes de dépendances entre les tâches de calcul résultent de l'emploi de données partagées passées en paramètres de ces tâches définissant la granularité de la parallélisation. Dans le cadre de la simulation de textiles, chaque tâche doit traiter les données contenues dans un ou deux

blocs de particules, les principaux calculs étant ceux relatifs aux forces, aux accélérations, et aux vitesses et positions des particules.

Les différentes tâches de la simulation ont été entièrement décrites au sein de ce chapitre en mettant en évidence leurs dépendances de données. Dans le cas du calcul des forces et dans ceux relatifs à l'emploi de la méthode d'intégration implicite, les tâches créées prennent deux blocs de particules en paramètres. Afin d'optimiser ces tâches, des tables de hachage ont été créées afin de connaître d'une part les paires de blocs ayant des interactions entre eux (afin de ne pas créer de tâches inutiles) et d'autre part, pour une paire de blocs donnée, la liste des paires de particules en interaction au sein de ces deux blocs (afin d'éviter de les rechercher systématiquement à l'intérieur des tâches, alors que ces interactions ne sont pas modifiées durant la simulation).

Par ailleurs, ATHAPASCAN autorise des accès aux données partagées en écriture cumulée, permettant à plusieurs processeurs de mettre à jour au même instant un bloc de donnée en employant une fonction associative et commutative des paramètres. Par exemple, dans le cas du calcul de la force exercée sur une particule donnée, les processeurs possédant ses voisines peuvent additionner en même temps les interactions produites par chacune d'entre elles afin d'obtenir au final la force appliquée sur cette particule.

Enfin dans le but d'augmenter l'efficacité de l'exécution parallèle, des algorithmes d'ordonnancement sont utilisés pour placer les données et les taches sur les processeurs en maximisant la localité des accès aux données et en équilibrant la charge de calculs au sein de la grappe.

Évaluation de performances 10

Ce chapitre présente l'analyse de performances de la parallélisation de la simulation de textiles. Les différents paramètres entrant en jeu dans l'obtention d'une exécution efficace vont être testés. Nous allons notamment voir les répercutions qu'ont la granularité, la stratégie de placement des données et l'ordonnancement des tâches sur les performances de l'application sur la grappe de machines.

10.1 Introduction

La parallélisation de notre simulation de textiles est basée sur un partitionnement de l'objet déformable en blocs de particules, la taille de ces blocs définissant la granularité de la parallélisation. Un certain nombre de paramètres entre ensuite en jeu dans l'obtention d'une exécution parallèle efficace : la taille des blocs issus du partitionnement, le nombre de processeurs impliqués, le placement des données partagées, l'ordonnancement des tâches sur les processeurs, etc... Nous allons dans ce chapitre évaluer l'impact de chacun de ces paramètres.

Afin de clarifier les différentes analyses, nous considérerons le cas d'un tissu régulier de taille 100×100 c'est-à-dire modélisé à l'aide de 10 000 particules. Ces particules seront partitionnées en quatre blocs possédant chacun 2 500 particules. La figure 10.1 illustre cet exemple en mettant en évidence le nombre d'interactions qui existent au sein de chacun des blocs et entre les blocs : 7301 particules en interaction au sein de chacun des 4 blocs, 99 entre les couples de blocs [0, 1], [0, 2], [1, 3] et [2, 3], et seulement 1 interaction pour le couple [0, 3]. Dans cet exemple il y a au total 29 601 particules en interaction et 9 interactions entre paires de blocs (5 interactions entre blocs différents et 4 interactions pour celles au sein des 4 blocs). Le nombre d'interactions entre particules

correspond ainsi à environ 3 fois le nombre de particules multipliant par 3 la taille de notre problème.

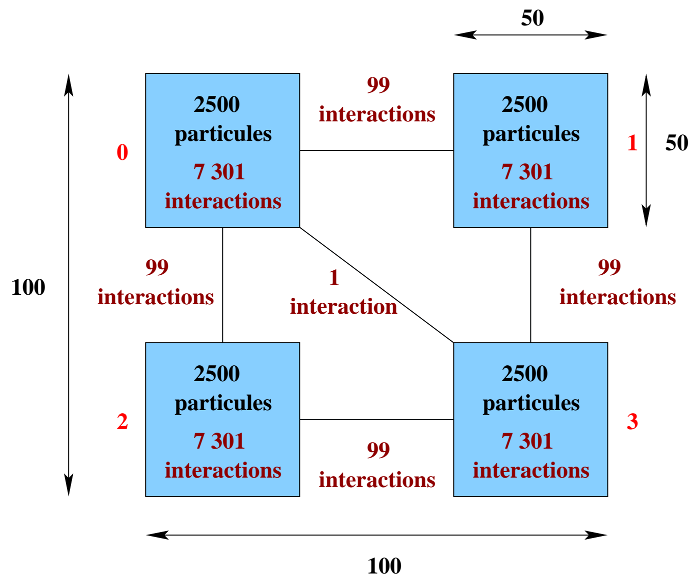


Figure 10.1 *Tissu régulier de taille* 100×100 *partitionné en 4 blocs identiques de 2 500 particules*

Si jamais ces 4 blocs se trouvent sur 4 processeurs différents le volume total de communications effectuées est alors de $5 \times 2500 = 12500$ particules dans le cas où l'ensemble du bloc est communiqué ou de $5 \times 50 = 250$ particules dans le cas où seule la frontière est communiquée d'un processeur à l'autre, soit un gain logique d'un facteur 50.

10.2 Complexité des algorithmes parallèles

10.2.1 Nombre de tâches créées

Les tableaux 10.1 et 10.2 présentent le nombre de tâches créées durant l'exécution de la simulation de textiles lors de l'emploi d'une méthode d'intégration explicite et lors de l'emploi de la méthode d'Euler implicite. Ces différentes tâches sont présentées en détails dans le chapitre 9. A ces tâches créées explicitement par l'utilisateur, il faut rajouter les tâches créées en interne par ATHAPASCAN correspondant à la création (`TaskNewAccess`) et à la destruction (`TaskDeleteFormat`) des données partagées, aux communications effectuées entre processeurs afin de transférer les données nécessaires à une tâche et aux vols de tâches entre processus. Les tâches `CopieInit` et `Alpha_Init`, que nous n'avions pas encore présentées, permettent de réinitialiser à chaque itération du Gradient Conjugué des variables accédées en écriture cumulée. Ces tâches ne sont instantiées qu'une seule fois par itération du GC.

	Type de la tâche	Nombre de tâches créées
Initialisation	`Init_Vect`	# Blocs
	`Init_List`	# Inter. entre paire de blocs
Force	`Force`	# Inter. entre paire de blocs
Accélération	`Accel`	# Blocs
Position et vitesse	`Integr_Expl`	# Blocs
Total		3 × # Blocs + 2 × # Inter. entre paire de blocs

TAB. 10.1 *Nombre de tâches créées durant la simulation parallèle : cas de l'emploi d'une méthode d'intégration explicite*

	Type de la tâche	Nombre de tâches créées
Initialisation	`Init_Vect`	# Blocs
	`Init_Vect_Impl`	# Blocs
	`Init_List`	# Inter. entre paire de blocs
	`Init_Matr`	# Blocs
Force et matrices	`Integr_Impl1`	# Inter. entre paire de blocs
Accélération	`Accel`	# Blocs
Vecteur B	`Integr_Impl2_Diag`	# Blocs
	`Integr_Impl2_NonDiag`	# Inter. entre paire de blocs
	`Integr_Impl2`	# Blocs
Gradient Conjugué (1 itération)	`Norme_Task`	# Blocs
	`Direction_Task`	# Blocs
	`ProdMatVect_W_Diag`	# Blocs
	`ProdMatVect_Dx_Dv_NonDiag`	# Inter. entre paire de blocs
	`CalculProdMatVect_Task`	# Blocs
	`Alpha_Task`	# Blocs
	`Solution_Residu_Task`	# Blocs
	`CopieInit`	1
	`Alpha_Init`	1
Vitesse	`Vitesse`	# Blocs
Position	`Position`	# Blocs
Total		(8 + 6 × # Iter. GC) × # Blocs + (3 + # Iter. GC) × # Inter. + 2

TAB. 10.2 *Nombre de tâches créées durant la simulation parallèle : cas de l'emploi d'une méthode d'intégration d'Euler implicite*

Si nous considérons l'exemple du tissu régulier de 10 000 particules partitionné en 4 blocs de 2 500 particules, il y a 9 interactions entre paires de blocs. Le nombre de tâches créées explicitement par l'utilisateur durant une itération de la simulation est ainsi de 30 dans le cas explicite et de 78 dans le cas implicite pour une seule itération du Gradient Conjugué. Par contre le nombre total de tâches réellement créées incluant celles effectuées par ATHAPASCAN est de 70 pour la méthode explicite et de 254 pour la méthode d'Euler implicite. Le nombre de tâches créées en interne n'est donc pas négligeable c'est pourquoi par la suite nous allons essayer de quantifier ce surcoût dû à l'emploi d'un langage parallèle de haut niveau.

Par ailleurs nous pouvons noter que la méthode implicite engendre plus de deux fois plus de tâches de calcul par rapport à la méthode explicite notamment dû à la résolution du système linéaire. Mais cette méthode d'Euler implicite permet l'emploi de pas de temps beaucoup plus important : elle nécessite moins d'itérations qu'une méthode explicite par rapport à un intervalle de temps donné. Ainsi si les pas de temps de la méthode implicite peuvent être deux fois supérieurs à ceux des méthodes explicites, le nombre de calculs engendrés redevient équivalent si une seule itération du Gradient Conjugué n'est nécessaire. Et comme en pratique la taille du pas de temps de la méthode d'Euler implicite n'a pas de limite, le désavantage de cette méthode dû au grand nombre de tâches peut être compensé grâce à un pas de temps conséquent.

De plus il ne faut pas oublier que les tâches créées durant l'initialisation sont par définition créées uniquement durant cette phase et ne doivent donc pas être comptabilisées pour chaque itération de la simulation.

10.2.2 Complexités pratiques

Dans le chapitre 8 nous avons vu que la complexité parallèle des calculs d'une simulation impliquant n particules était en $\mathcal{O}(\frac{n}{p})$ avec p le nombre de processeurs. En effet le coût pour une boîte est alors en $\mathcal{O}(\frac{n}{k})$ avec k le nombre de boîtes et au total il y a $\frac{k}{p}$ boîtes par processeur.

Mais il est possible d'affiner un peu plus ce coût théorique en faisant intervenir les coûts relatifs à la création des tâches et à leur exécution. Le coût relatif aux tâches traitant un seul bloc à la fois (ex : `Accel`) est en

$$\mathcal{O}\left(\frac{n}{k} \times C_0 + C_1\right) \times \frac{k}{p}\, C_2$$

et le coût pour les tâches traitant une paire de blocs (ex : `Force`) est en

$$\mathcal{O}\left(\frac{P_{Inter}}{\#Inter.} \times C_0 + C_1\right) \times \#Inter.\, C_2$$

avec

- P_{Inter} le nombre de particules en interaction,

- # Inter. le nombre d'interactions entre paires de blocs,
- C_0 le coût de traitement d'une particule,
- C_1 le coût de traitement de la boucle au sein de la tâche
- et C_2 le coût de création d'une tâche avec normalement $C_0 >> C_2 > C_1$.

Nous allons par la suite essayer de quantifier chacun de ces coûts.

10.2.3 Temps d'exécution des tâches

Les tableaux 10.3 et 10.4 présentent les temps de calcul séquentiels de l'ensemble des tâches nécessaires à la méthode de Leapfrog et d'Euler implicite pour simuler le comportement de 10 000 particules décomposées en 4 blocs de même taille sur 4 itérations avec une seule itération pour la méthode du Gradient Conjugué dans le cas implicite. Ces tests ont été réalisés sur un bi-processeur cadencé à 2,4 Go avec 1,5 Go de RAM du *cluster IDPOT* (projet cluster DELL ID-IMAG). Nous rappelons également que dans cet exemple il y a 9 interactions entre paire de blocs.

Nous pouvons observer que le nombre d'appels de chacune des tâches correspond bien au nombre théorique présenté dans les tableaux 10.1 et 10.2 en n'oubliant pas que nos observations relatent dans cette section de quatre itérations de la simulation.

Les tâches les plus coûteuses en temps de calcul sont la tâche `Force` pour la méthode explicite et la tâche `Integr_Impl1` pour la méthode implicite qui est équivalente à la tâche `Force`. Ce résultat est tout à fait normal puisque ces deux tâches ont à traiter l'ensemble des interactions présentes dans la simulation et que ce nombre est environ trois fois plus important que le nombre de particules. En implicite les tâches `Integr_Impl2_NonDiag` et `ProdMatVect_Dx_Dv_NonDiag` doivent également traiter le même nombre d'opérations mais celles-ci sont plus rapides que celles faites durant le calcul des forces car elle ne représentent grossièrement qu'une multiplication et une somme.

Nous pouvons également noter qu'il y a un nombre important de tâches créées par ATHAPASCAN pour la création et à la destruction des données partagées. Mais ces tâches (`TaskNewAccess` et `TaskDeleteFormat`) ayant des temps d'exécution presque négligeables ne perturbent pas les autres tâches.

Par ailleurs nous pouvons de nouveau remarquer que la parallélisation d'une application de simulation de textiles implique un grain très fin de calcul puisque les temps séquentiels sont déjà relativement faibles. Dans notre cas ils sont d'autant plus faibles que notre modèle physique est pour le moment assez simple. La complexification de ce modèle par l'ajout de ressorts de flexions par exemple augmentera légèrement ces temps.

Type de la tâche	% Temps séq.	Temps total (s)	Temps moyen (s)	# Appels
`Force`	84.05	0.063787	0.001771	36
`Accel`	4.692	0.003560	0.000222	16
`Init_List`	3.699	0.002807	0.000311	9
`Integr_Expl`	4.691	0.003559	0.000221	16
`Init_Vect`	2.866	0.002175	0.000543	4
`TaskNewAccess`	0.002066	1.568e-06	4.021e-08	39
`TaskDeleteFormat`	0.001057	8.025e-07	8.025e-07	1

TAB. 10.3 *Tableau des temps d'exécution de chacune des tâches engendrées lors de la simulation avec l'emploi de la méthode d'intégration de Leapfrog*

Type de la tâche	% Temps séq.	Temps total (s)	Temps moyen (s)	# Appels
`Integr_Impl1`	64.66	0.527787	0.014660	36
`ProdMatVect_Dx_Dv_NonDiag`	15.88	0.129646	0.003601	36
`Integr_Impl2_NonDiag`	9.54	0.077878	0.002163	36
`ProdMatVect_W_Diag`	1.695	0.013836	0.000864	16
`Integr_Impl2_Diag`	1.666	0.013600	0.000850	16
`Init_Matr`	1.435	0.011717	0.002929	4
`Vitesse`	1.107	0.009032	0.000564	16
`Init_Vect_Impl`	0.6828	0.005574	0.001393	4
`Solution_Residu_Task`	0.6481	0.005290	0.000330	16
`CalculProdMatVect_Task`	0.446	0.003640	0.000227	16
`Init_List`	0.4181	0.003412	0.000379	9
`Integr_Impl2`	0.3944	0.003219	0.000201	16
`Accel`	0.3538	0.002887	0.000180	16
`Init_Vect`	0.2574	0.002101	0.000525	4
`Direction_Task`	0.2317	0.001891	0.000118	16
`Position`	0.1813	0.001480	9.251e-05	16
`Alpha_Task`	0.1721	0.001404	8.780e-05	16
`Norme_Task`	0.1679	0.001370	8.564e-05	16
`CopieInit`	0.06189	0.000505	0.000126	4
`TaskDeleteFormat`	0.001238	1.010e-05	2.463e-07	41
`Alpha_Init`	0.0005889	4.806e-06	1.201e-06	4
`TaskNewAccess`	0.0005788	4.725e-06	3.970e-08	119

TAB. 10.4 *Tableau des temps d'exécution de chacune des tâches engendrées lors de la simulation avec l'emploi de la méthode d'intégration d'Euler implicite*

10.2.4 Potentiel de parallélisme

Les tableaux 10.5 et 10.6 présentent les temps T_1 et T_∞ de l'exécution dans le cas explicite et implicite. T_1 représente la somme des temps d'exécution de chacune des tâches et T_∞ le temps sur un nombre infini de processeurs soit le chemin critique du graphe de flots de données associé à l'application. Ces tests ont également été réalisés sur un noeud du cluster DELL présenté précédemment.

	Temps d'exécution	**# Tâches**
T_1	0.0755682 s	121
T_∞	0.0170474 s	13
T_1 / T_∞	4.43282	

TAB. 10.5 *Temps T_1 et T_∞ de l'exécution dans le cas explicite*

	Temps d'exécution	**# Tâches**
T_1	0.812066 s	473
T_∞	0.136727 s	57
T_1 / T_∞	5.93932	

TAB. 10.6 *Temps T_1 et T_∞ de l'exécution dans le cas implicite*

Dans le cas explicite, il y a au total 121 tâches à exécuter dont 13 tâches constituent le chemin critique du graphe de flots de données : ces tâches ne peuvent être exécutées en concurrence. Le potentiel de parallélisme qui n'est autre que le rapport de T_1 sur T_∞ est logiquement proche de 4 le nombre de blocs de la découpe.

Dans le cas implicite, il y a 473 tâches à exécuter dont 57 se trouvent sur le chemin critique du graphe de flots de données. Le potentiel de parallélisme est supérieur à 4 car la partie concernant la résolution du Gradient Conjugué engendre un grand nombre de tâches prenant en paramètres les données partagées en écriture cumulée : plusieurs tâches sont exécutées en parallèle pour effectuer le traitement d'un bloc donné et non une seule. C'est pourquoi potentiellement cette application a une ressource de parallélisme supérieure au nombre de blocs.

Ces résultats confirment que le nombre de blocs créés permet d'augmenter les sources de parallélisme de l'application. Mais il ne faut pas tomber dans l'excès car nous verrons par la suite que les temps de communications sont proches des temps de calcul et plus il y a de blocs plus le volume de communications augmente.

10.3 Mémoire utilisée

Soit n le nombre de particules dans la simulation (10 000 dans notre exemple). Les propriétés des particules (position, vitesse, accélération, force) sont définies dans les trois directions de l'espace (x, y, z). Chacune des caractéristiques des particules nécessitent alors 3 unité de stockage (float ou double). Par ailleurs pour une paire de blocs, élément de la structure `list`, la liste des couples de particules en interaction sont stockées dans la structure `list_part` : chaque élément de cette liste est ainsi une structure comportant l'identification des deux particules concernées ainsi que les propriétés physiques du ressort les reliant (raideur, longueur au repos, amortissement, facteur d'amortissement) soit un total de 6 éléments. Dans le cadre de la méthode d'Euler implicite il faut rajouter à ces structures les matrices de contributions des forces. Leurs éléments sont des matrices diagonales de taille 3×3 : ces matrices étant diagonales il est possible de ne stocker que 6 éléments sur les 9.

Les tableaux 10.8 et 10.7 présentent le coût en mémoire de chacune des structures de données en implicite et en explicite avec "# Inter." le nombre d'interaction entre paires de blocs (9 dans notre exemple de départ) et "P_{inter}" le nombre total de particules en interaction (29 601 pour notre exemple).

Nom de la structure	**Mémoire employée**	**Mémoire employée pour 10 000 particules**	
`Force[]`	$3 \times n \times 4 = 12$ n	120 Ko	(2,5 %)
`Position[]`	$3 \times n \times 4 = 12$ n	120 Ko	(2,5 %)
`Vitesse[]`	$3 \times n \times 4 = 12$ n	120 Ko	(2,5 %)
`Accel[]`	$3 \times n \times 4 = 12$ n	120 Ko	(2,5 %)
`Masse[]`	$1 \times n \times 4 = 4$ n	40 Ko	(0,83 %)
`Matrice_df_dx_diag[]`	$6 \times n \times 4 = 24$ n	240 Ko	(5 %)
`Matrice_df_dx[][]`	$6 \times 2 \times P_{inter} \times 4 = 48\ P_{inter}$	1.42 Mo	(**29,6 %**)
`Matrice_df_dv_diag[]`	$6 \times n \times 4 = 24$ n	240 Ko	(5 %)
`Matrice_df_dv[][]`	$6 \times 2 \times P_{inter} \times 4 = 48\ P_{inter}$	1.42 Mo	(**29,6 %**)
`B[]`	$3 \times n \times 4 = 12$ n	120 Ko	(2,5 %)
`X[]`	$3 \times n \times 4 = 12$ n	120 Ko	(2,5 %)
`list`	$2 \times$ # Inter. $\times 4 = 8$ # Inter.	72 o	(0,001 %)
`list_part`	$6 \times P_{inter} \times 4 = 24\ P_{inter}$	710 Ko	(**14,8 %**)
Total	124 n + 120 P_{inter} + 8 # Inter.	4,317 Mo	

TAB. 10.7 *Mémoire utilisée dans chacune des structures de données de la simulation de textiles : cas de l'emploi de la méthode d'intégration d'Euler implicite*

Nom de la structure	**Mémoire employée**	**Mémoire employée pour 10 000 particules**	
`Force[]`	$3 \times n \times 4 = 12$ n	120 Ko	(9,75 %)
`Position[]`	$3 \times n \times 4 = 12$ n	120 Ko	(9,75 %)
`Vitesse[]`	$3 \times n \times 4 = 12$ n	120 Ko	(9,75 %)
`Accel[]`	$3 \times n \times 4 = 12$ n	120 Ko	(9,75 %)
`Masse[]`	$1 \times n \times 4 = 4$ n	40 Ko	(3,25 %)
`list`	$2 \times$ # Inter. $\times 4 = 8$ # Inter.	72 o	(0,006 %)
`list_part`	$6 \times P_{inter} \times 4 = 24\ P_{inter}$	710 Ko	(**57,72 %**)
Total	52 n + 8 # Inter. + 24 P_{inter}	1,23 Mo	

TAB. 10.8 *Mémoire utilisée dans chacune des structures de données de la simulation de textiles : cas de l'emploi d'une méthode d'intégration explicite*

Les structures les plus coûteuses à stocker sont donc la liste des particules en interaction dans les deux cas (explicite et implicite) et les matrices des contributions des forces dans le cas implicite. La taille de ces structures dépend en effet du nombre de particules en interaction : le nombre d'éléments contenus dans les matrices correspondant à deux fois ce nombre d'interactions. Or ce nombre d'interactions est 3 fois plus important que le nombre de particules : il y a ainsi un facteur 36 de différence entre le nombre de particules et le nombre d'éléments dans ces matrices. Par ailleurs pour chaque couple de particules en interaction, les propriétés du ressort les reliant sont stockées impliquant une structure de donnée au final assez conséquente.

Une optimisation immédiate en agglomérant ces données permettrait de réduire ce coût. Par ailleurs à l'heure actuelle la totalité des éléments non nuls des matrices des contributions des forces sont stockés. Or ces matrices étant symétriques, il serait possible de ne stocker que la moitié de ces matrices en faisant en sorte que les calculs tels que les produits de ces matrices par un vecteur quelconque se fassent correctement et en prenant garde que ce gain en mémoire ne devienne pas problématique au niveau de la parallélisation en rajoutant des dépendances de données.

10.4 Influence de la granularité

La parallélisation de la simulation de textiles repose sur le partitionnement du tissu en sous ensembles de particules. A plusieurs reprises nous avons signalé que le choix de cette découpe, c'est-à-dire la taille des blocs choisi pour un nombre de particules donné, devait être effectué judicieusement pour obtenir de bonnes performances. Pour mettre en évidence ce fait nous allons faire varier pour un problème donné la taille des blocs de par-

ticules et ainsi le nombre de tâches à traiter. Cette expérimentation est réalisée dans le cas de l'emploi de la méthode d'intégration de Leapfrog ainsi que dans le cas de la méthode d'Euler implicite.

Il faut savoir par ailleurs qu'une application programmée à l'aide de l'environnement de programmation parallèle ATHAPASCAN peut être exécutée aussi bien en séquentiel, en SMP ou en distribuée sans aucune modification du code. Les temps présentés dans cette section sont relatifs à une exécution séquentielle de l'application mais gérée par ATHAPASCAN : ces temps comprennent ainsi le surcoût lié à l'emploi d'ATHAPASCAN.

Dans le cas de l'utilisation de la méthode de Leapfrog, la figure 10.2 présente l'évolution du temps d'exécution de la simulation sur 1 noeud ainsi que le nombre de tâches créées en fonction du nombre de blocs issus du partitionnement du tissu. Les deux courbes de cette figure représentent 5 itérations de la simulation de 10000 particules exécutée sur un noeud du cluster DELL.

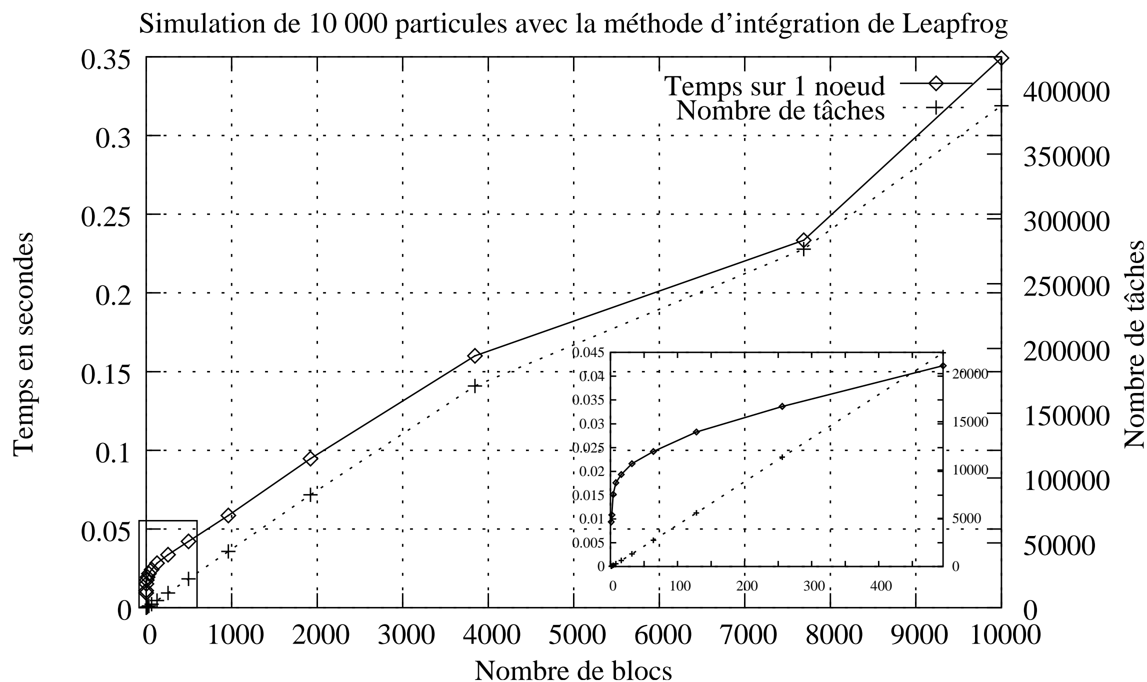


Figure 10.2 *Simulation de 10 000 particules avec la méthode d'intégration de Leapfrog : influence de la découpe sur le nombre de tâches créées et sur le temps d'exécution sur 1 noeud*

La figure 10.3 présente ensuite cette même évolution dans le cas de l'utilisation de la méthode d'Euler implicite. Les deux courbes de cette figure représentent 5 itérations de la simulation de 10000 particules avec une seule itération de la méthode du Gradient Conjugué également exécutée sur un noeud du cluster DELL.

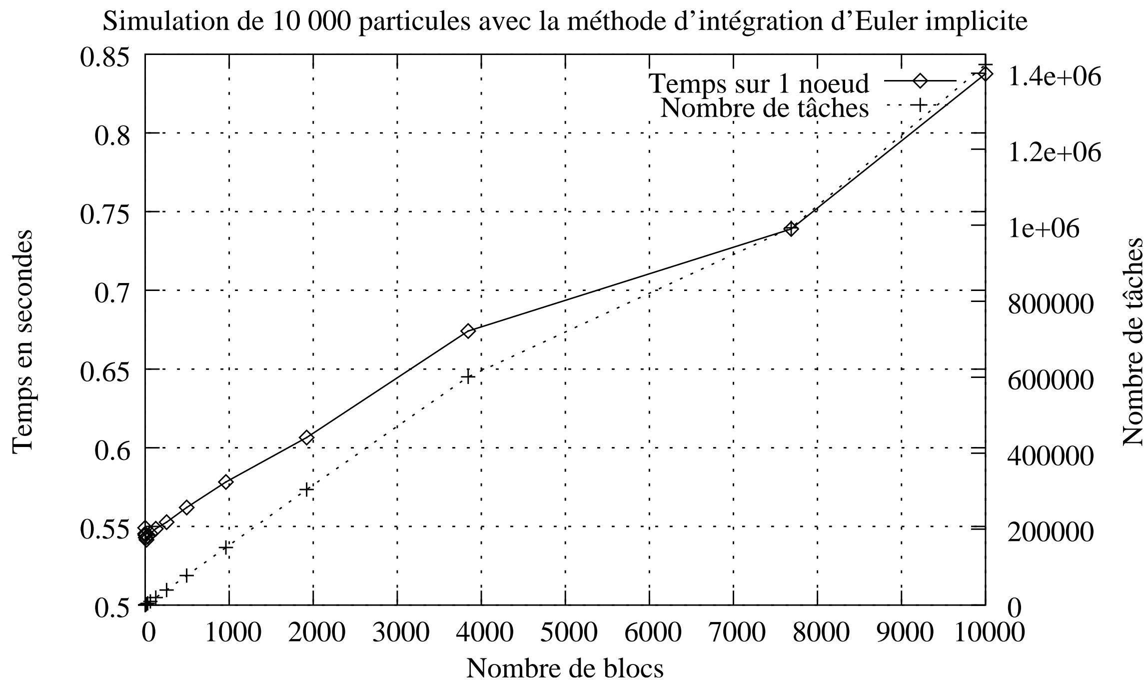


Figure 10.3 *Simulation de 10 000 particules avec la méthode d'intégration d'Euler implicite : influence de la découpe sur le nombre de tâches créées et sur le temps d'exécution sur 1 noeud*

Les deux courbes de chacune des deux figures ont des formes similaires mettant en évidence le fait que plus il y a de tâches créées dans l'application plus le temps d'exécution en séquentiel est important : chaque tâche créée possède un coût au niveau de sa gestion par ATHAPASCAN et le fait d'instantier un grand nombre de tâches augmente ainsi ce surcoût.

Les points finaux de ces courbes représentent une découpe en 10000 blocs soit seulement une particule dans chacun des blocs. Le surcoût est alors maximal avec 387210 tâches créées au lieu de 28 dans le cas où l'application ne comporte qu'un seul bloc en explicite et 1423242 tâches au lieu de 134 en implicite. Les temps d'exécution passent alors de $0.009384\ s$ à $0.349259\ s$ en explicite et de $0.549095\ s$ à $0.837552\ s$ en implicite.

Il est ainsi très important de trouver le nombre de blocs adapté à chaque simulation. A l'heure actuelle la granularité est choisie assez expérimentalement en prenant en compte le nombre de processeurs visés ainsi que le nombre de particules de l'application : si le tissu est découpé en 4 sous-blocs, les premiers tests de performances observeront les résultats sur au moins 4 processeurs. De plus la taille des blocs est souvent corrélée à la racine du nombre de particules : pour un tissu de 10000 particules, les tailles de bloc intéressantes sont en pratique multiples de 100.

Mais à terme, il serait intéressant d'essayer d'implanter une méthode fiable permet-

tant, à partir d'un problème donné, de trouver la meilleure granularité. Pour cela, nous avions vu que l'utilisation de bibliothèques de partitionnement de graphe telles que Scotch [94, 93] ou encore Métis [69, 70, 110] permettrait d'obtenir cette granularité optimale avec en prime une répartition optimale des particules dans chacun des blocs.

10.5 Influence de l'ordonnancement

Le choix de l'ordonnancement employé pour placer les données sur les processeurs et ensuite pour distribuer les tâches associées sur la grappe a un rôle déterminant dans l'obtention de performances correctes. Si ces placements sont effectués non judicieusement ils peuvent engendrer un grand nombre de communications inutiles annulant les gains obtenus au niveau des calculs.

C'est pourquoi dans cette partie nous allons tout d'abord présenter pour les cas explicite et implicite les graphes de flots de données (GFD) associés à la simulation en identifiant clairement chacune des tâches impliquées. Puis nous verrons les différents placements possibles de ce graphe en fonction de l'ordonnancement choisi.

Nous allons ainsi comparer deux stratégies d'ordonnancement. La première est celle obtenue par l'emploi de la bibliothèque Scotch [94, 93]. Et la seconde appelée Cyclic repose sur une distribution Cyclic des données avec un placement des tâches selon la règle *OwnerComputeRule* (cf. chapitre 9, section 9.3). Nous observerons notamment leur répercution sur le placement du graphe de flots de données (placement des tâches et des données sur les processeurs) ainsi que sur le temps de l'exécution des tâches.

10.5.1 Placement du GFD explicite

Considérons dans un premier temps le cas de l'emploi de méthode d'intégration de Stöermer-Verlet/Leapfrog. La figure 10.4 présente en effet le graphe de flots de données correspondant à une itération de la simulation de 10000 particules non partitionnées lors de l'emploi de cette méthode d'intégration explicite. Ce GFD est donc relatif au traitement d'un seul sous-ensemble de particules. Nous retrouvons sur ce graphe l'ensemble des tâches créées répertoriées dans la section 10.2.1. Les tâches non étiquetées sont celles créées en interne par ATHAPASCAN (`TaskNewAccess` et `TaskDeleteFormat`). La tâche `EnvoiPosition` figure également dans ce graphe : elle permet l'envoi des positions au module de visualisation (détaillé dans le chapitre suivant).

Puis la figure 10.5 présente le graphe de flots de données correspondant à une itération de la simulation de 10000 particules décomposées en 4 blocs de 2500 particules. Cette fois-ci la tâche `EnvoiPosition` ne figure pas dans ce graphe. Il faut noter que ce graphe est le GFD original de l'application : aucun placement n'a encore été établi pour

chacune des tâches et chacune des données partagées. Nous retrouvons bien au sein de cette figure la structure du graphe précédent pour un bloc : les 4 blocs engendrent les mêmes dépendances de données.

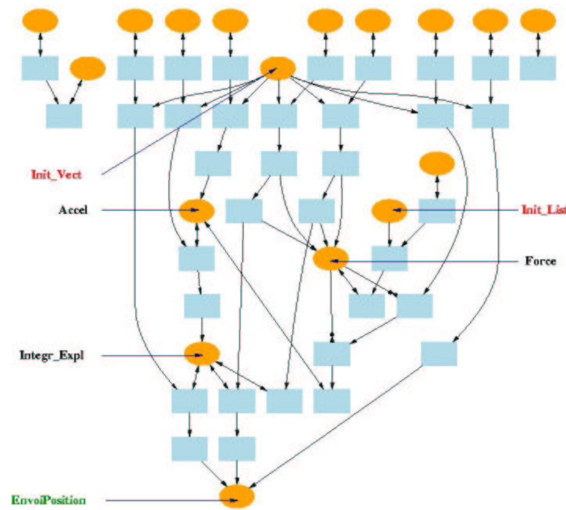


Figure 10.4 *Graphe de flots de données associé à la simulation lors de l'emploi d'une méthode d'intégration explicite*

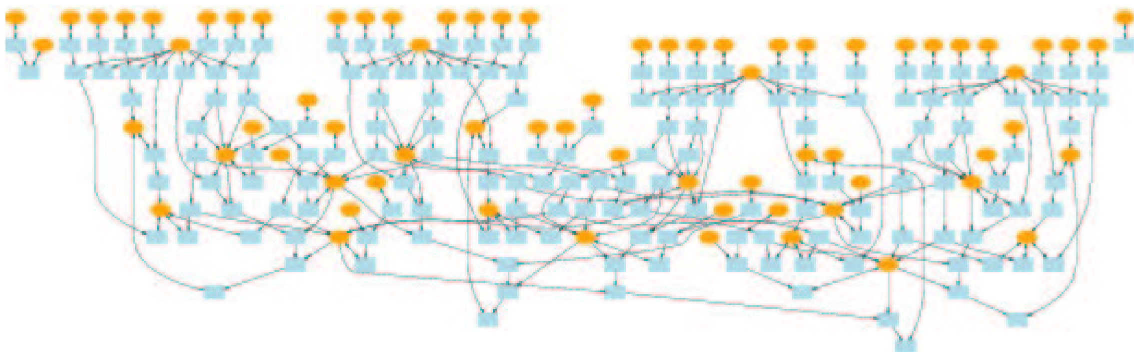


Figure 10.5 *Graphe de flots de données associé à la simulation partitionnée en 4 sous-ensembles de particules lors de l'emploi d'une méthode d'intégration explicite*

Ensuite la figure 10.6 présente le placement du graphe de flots de données effectué en utilisant les stratégies d'ordonnancement Scotch et Cyclic sur 2 processeurs. Les couleurs des noeuds du graphe représentent les placements de ces tâches sur les processeurs. Ce graphe de flots de données a été réalisée pour cinq itérations de la simulation décomposée en 4 blocs.

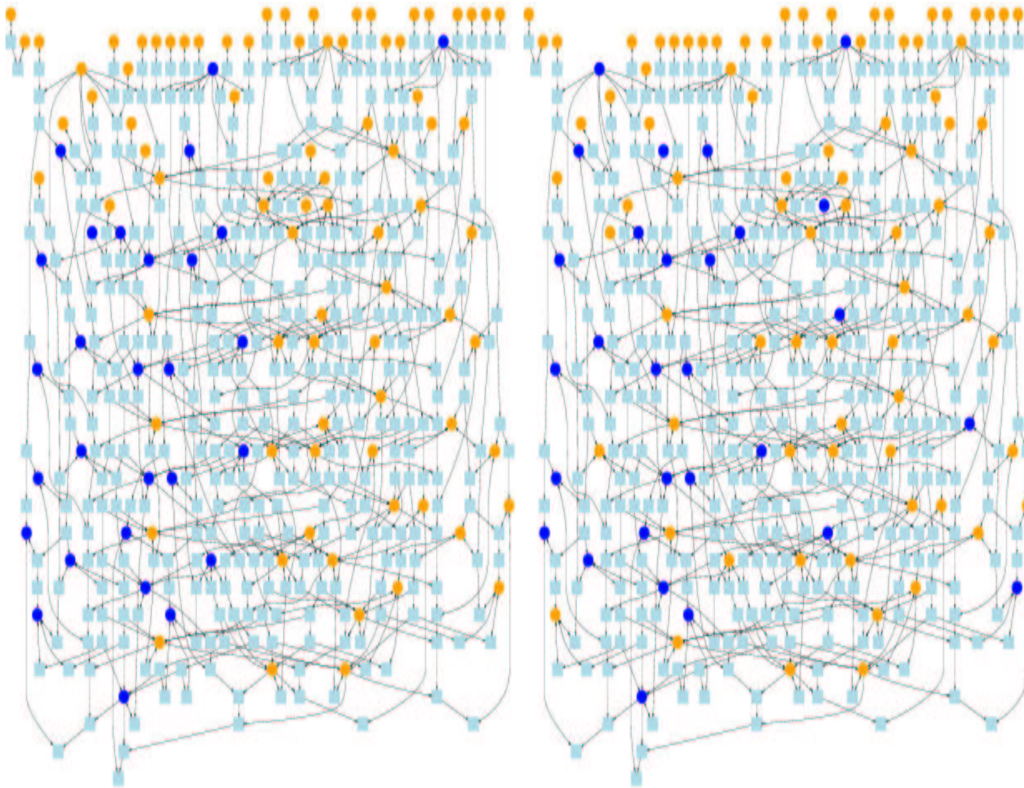


Figure 10.6 *Placement Scotch (à gauche) et Cyclic (à droite) du graphe de flots de données explicite sur 5 itérations pour un partitionnement en 4 sous-ensembles de particules sur 2 processeurs*

Le tableau 10.9 présente les données chiffrées du nombre de tâches de calculs et de communications placées sur chacun des 2 processeurs avec une stratégie d'ordonnancement Scotch (en haut) et Cyclic (en bas).

Numéro du noeud	# de tâches exécutées	# de tâches de communication
0	194	17
1	126	35

Numéro du noeud	# de tâches exécutées	# de tâches de communication
0	284	49
1	213	62

TAB. 10.9 *Nombre de tâches du GFD explicite placées sur 2 processeurs lors de l'emploi des stratégies Scotch (en haut) et Cyclic (en bas) pour 5 itérations de la simulation partitionnée en 4 sous-ensembles de particules*

Puis la figure 10.7 présente le placement du même graphe effectué cette fois-ci sur 4 processeurs.

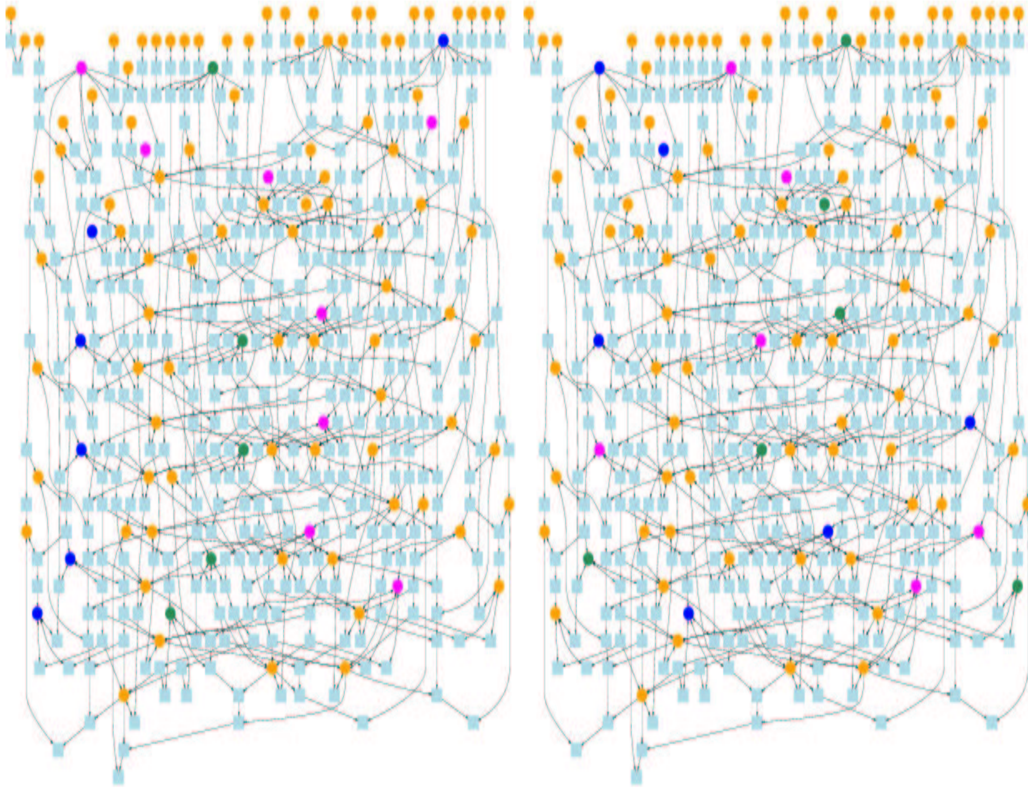


Figure 10.7 *Placement Scotch (à gauche) et Cyclic (à droite) du graphe de flots de données explicite sur 5 itérations pour un partitionnement en 4 sous-ensembles de particules sur 4 processeurs*

Le tableau 10.10 présente les données chiffrées du nombre de tâches de calculs placées sur chacun des 4 processeurs avec une stratégie d'ordonnancement Cyclic ou Scotch.

Numéro du noeud	# de tâches exécutées (Scotch)	# de tâches exécutées (Cyclic)
0	218	245
1	42	69
2	48	69
3	39	69

TAB. 10.10 *Nombre de tâches du GFD explicite placées sur 4 processeurs lors de l'emploi des stratégies Scotch et Cyclic pour 5 itérations de la simulation partitionnée en 4 sous-ensembles de particules*

Les figures 10.8 et 10.9 présentent le partitionnement du graphe de flots de données effectués par Scotch pour respectivement un placement sur 2 et 4 processeurs (cf. chapitre 9, section 9.3).

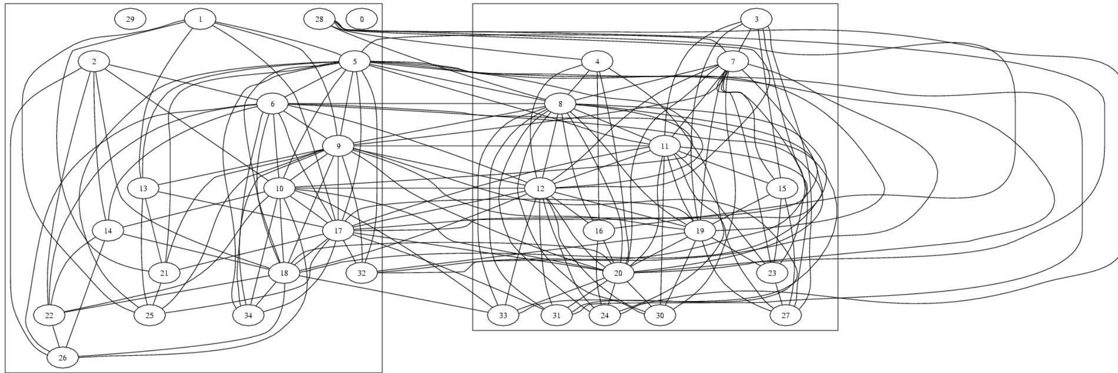


Figure 10.8 *Partitionnement Scotch du graphe de flots de données explicite sur 5 itérations pour un découpage en 4 sous-ensembles de particules sur 2 processeurs*

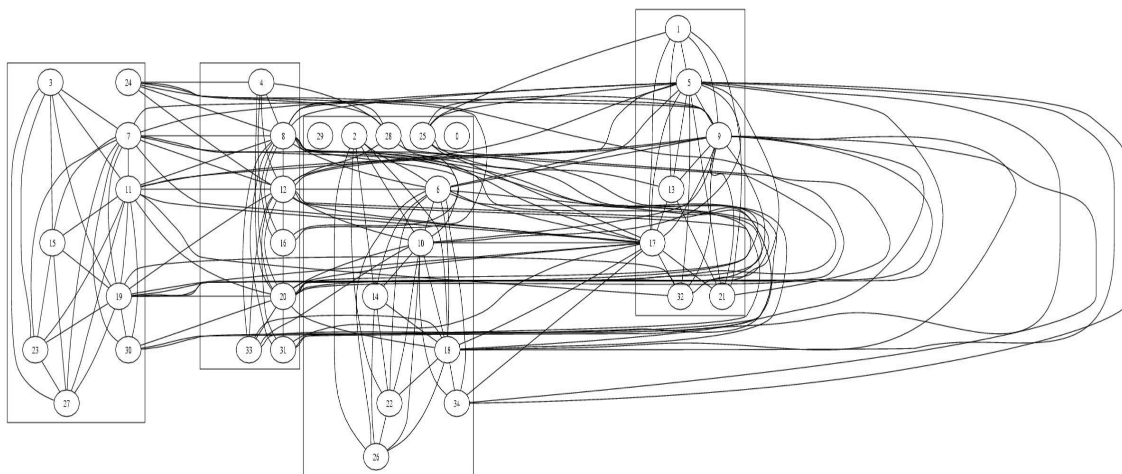


Figure 10.9 *Partitionnement Scotch du graphe de flots de données explicite sur 5 itérations pour un découpage en 4 sous-ensembles de particules sur 4 processeurs*

Dans ces exemples le placement Cyclic est légèrement meilleur que celui proposé par Scotch. Il faut dire que pour optimiser l'emploi de la bibliothèque Scotch il est possible d'attacher des coûts sur chacune des tâches et qu'à l'heure actuelle l'application de simulation de textiles ne donne pas ces coûts. Mais ce rajout peut être réalisé très rapidement. Par ailleurs, ces placements ont été réalisés sur un faible nombre d'itérations afin de rendre les graphes de flots de données lisibles : ceci ne facilite pas forcément la tâche de l'ordonnanceur.

10.5.2 Placement du GFD implicite

La figure 10.10 présente le GFD correspondant à une itération de la simulation de 10000 particules non partitionnées lors de l'emploi de la méthode d'intégration d'Euler implicite avec une seule itération utilisée dans la méthode du Gradient Conjugué. Nous retrouvons sur ce graphe l'ensemble des tâches créées répertoriées dans la section 10.2.1 ainsi que la tâche `EnvoiPosition`. Les tâches non étiquetées sont celles créées en interne par ATHAPASCAN (`TaskNewAccess` et `TaskDeleteFormat`).

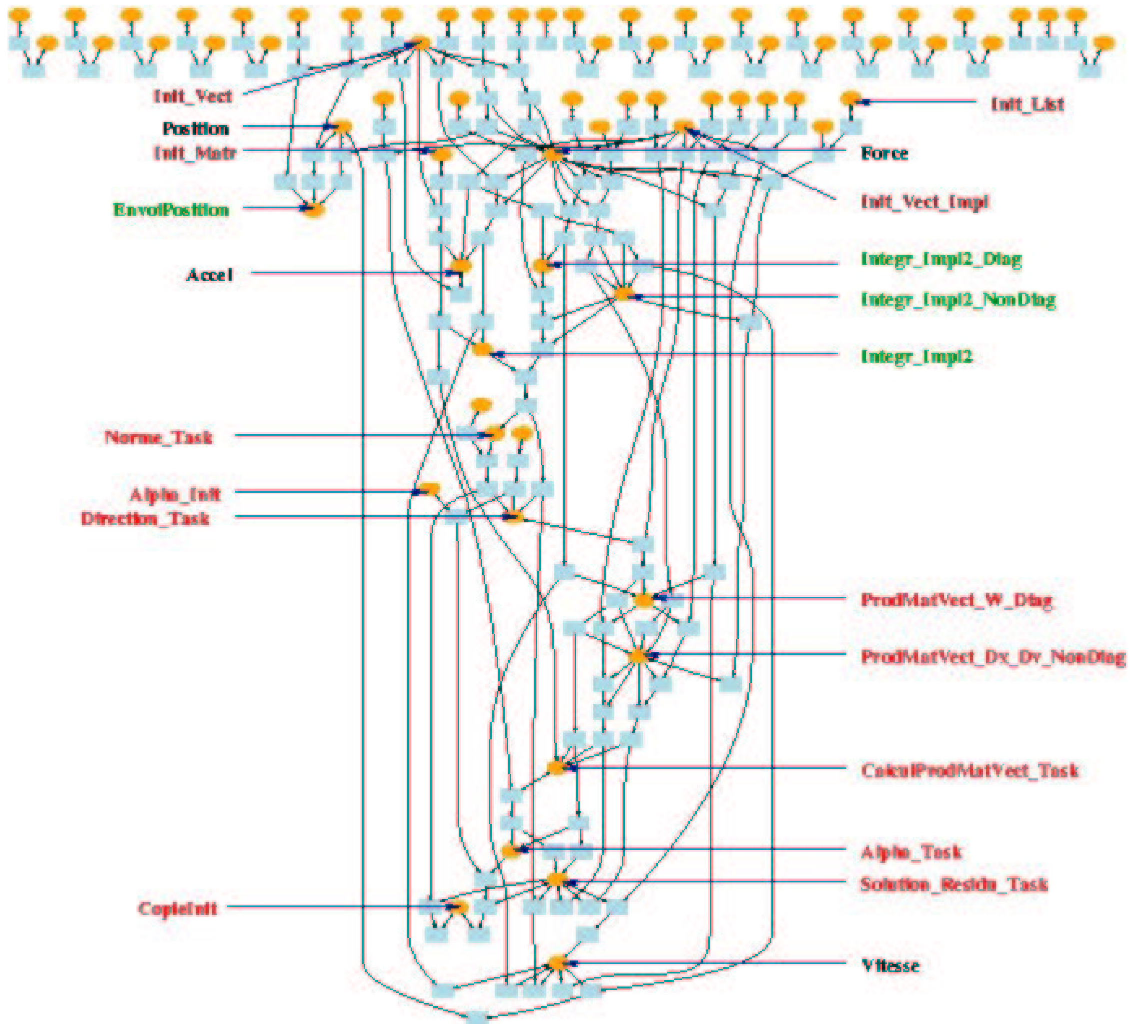


Figure 10.10 *Graphe de flots de données associé à la simulation lors de l'emploi de la méthode d'intégration d'Euler implicite*

Puis la figure 10.11 présente le GFD de la simulation de 10000 particules décomposées en 4 blocs de 2500 particules. Cette fois-ci la tâche `EnvoiPosition` ne figure pas dans ce graphe.

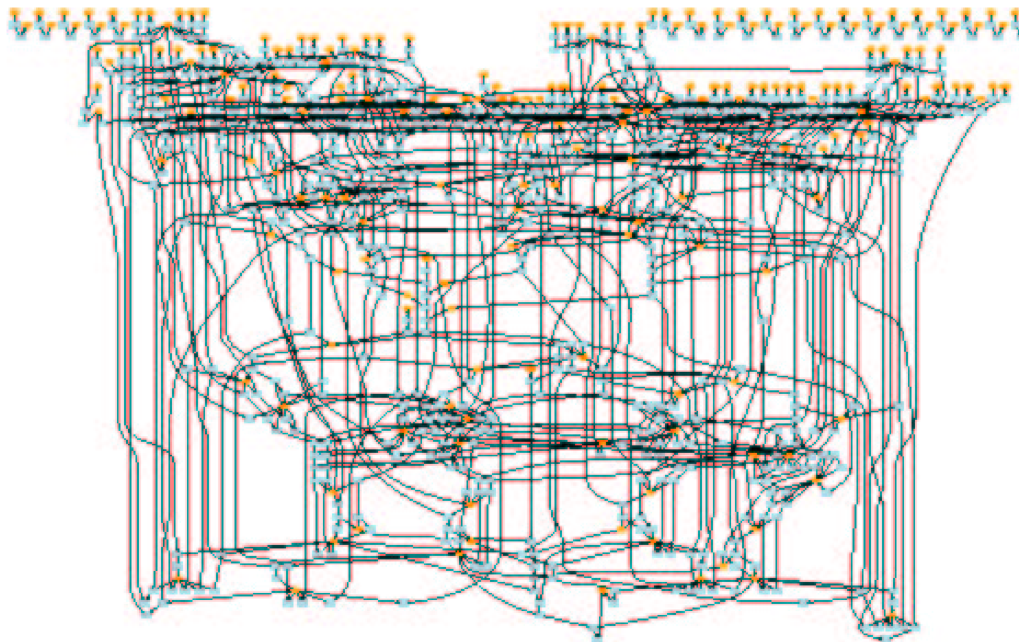


Figure 10.11 *Graphe de flots de données associé à la simulation partitionnée en 4 sous-ensembles de particules lors de l'emploi de la méthode d'Euler implicite*

De même que pour le graphe explicite, ce graphe représente le graphe de flots de données original de l'application sans aucun placement dans lequel nous retrouvons pour chacun des blocs les mêmes dépendances.

Le tableau 10.11 présente ensuite le nombre de tâches de calculs et de communications issues de ce GFD placées sur chacun des 2 processeurs avec une stratégie d'ordonnancement Scotch (en haut) ou Cyclic (en bas).

Numéro du noeud	# de tâches exécutées	# de tâches de communication
0	553	79
1	340	60

Numéro du noeud	# de tâches exécutées	# de tâches de communication
0	729	132
1	558	128

TAB. 10.11 *Nombre de tâches du GFD implicite placées sur 2 processeurs lors de l'emploi des stratégies Scotch (en haut) et Cyclic (en bas) pour 2 itérations de la simulation partitionnée en 4 sous-ensembles de particules*

Ensuite la figure 10.12 présente le placement du GFD effectuées en utilisant les stra-

tégies d'ordonnancement Scotch et Cyclic sur 2 processeurs. Ce graphe a été réalisé pour deux itérations de la simulation décomposée en 4 blocs avec une seule itération du GC. Les tâches relatives à l'initialisation ont été supprimées afin de clarifier ce graphe.

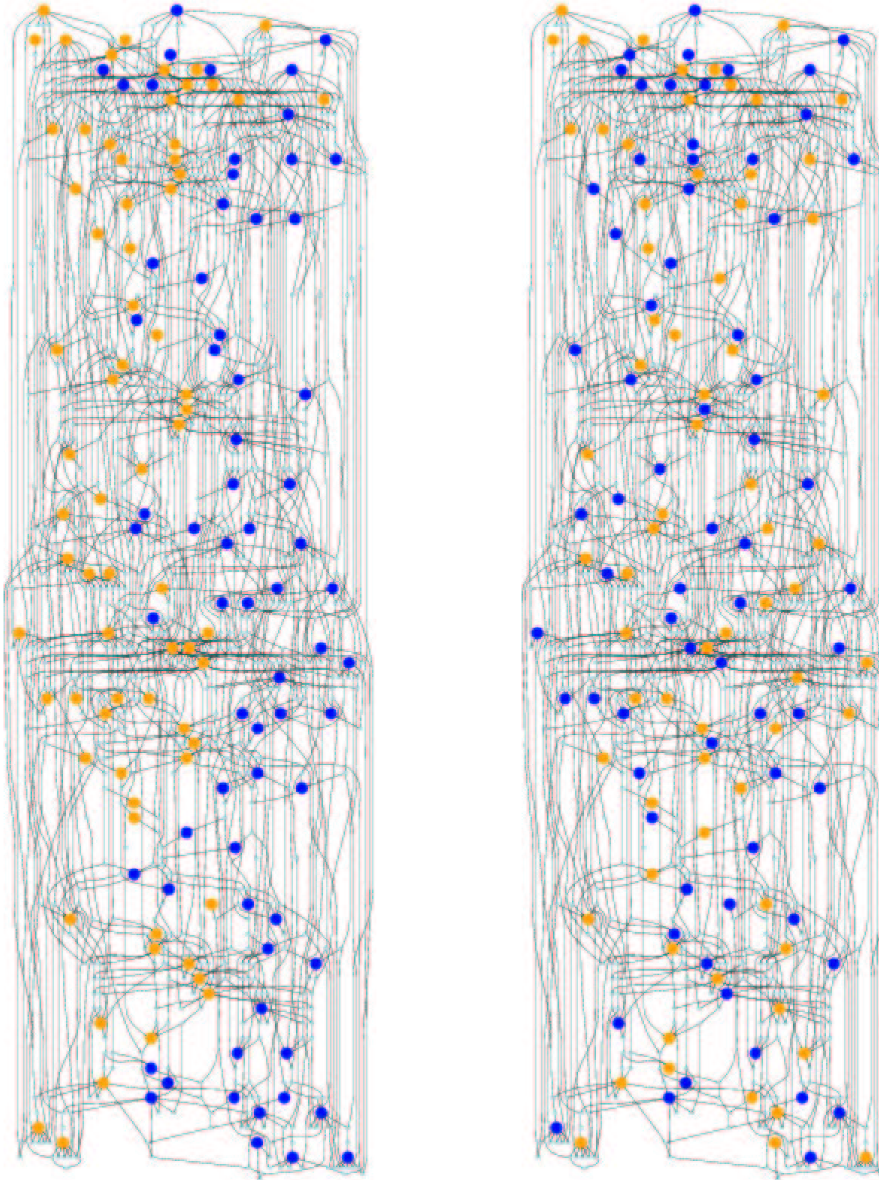


Figure 10.12 *Placement Scotch (à gauche) et Cyclic (à droite) du GFD implicite sur 2 itérations avec 1 itération du GC pour un partitionnement en 4 sous-ensembles de particules sur 2 processeurs*

Puis la figure 10.13 présente les mêmes stratégies appliquées sur le graphe de flots de données mais cette fois-ci pour 4 processeurs.

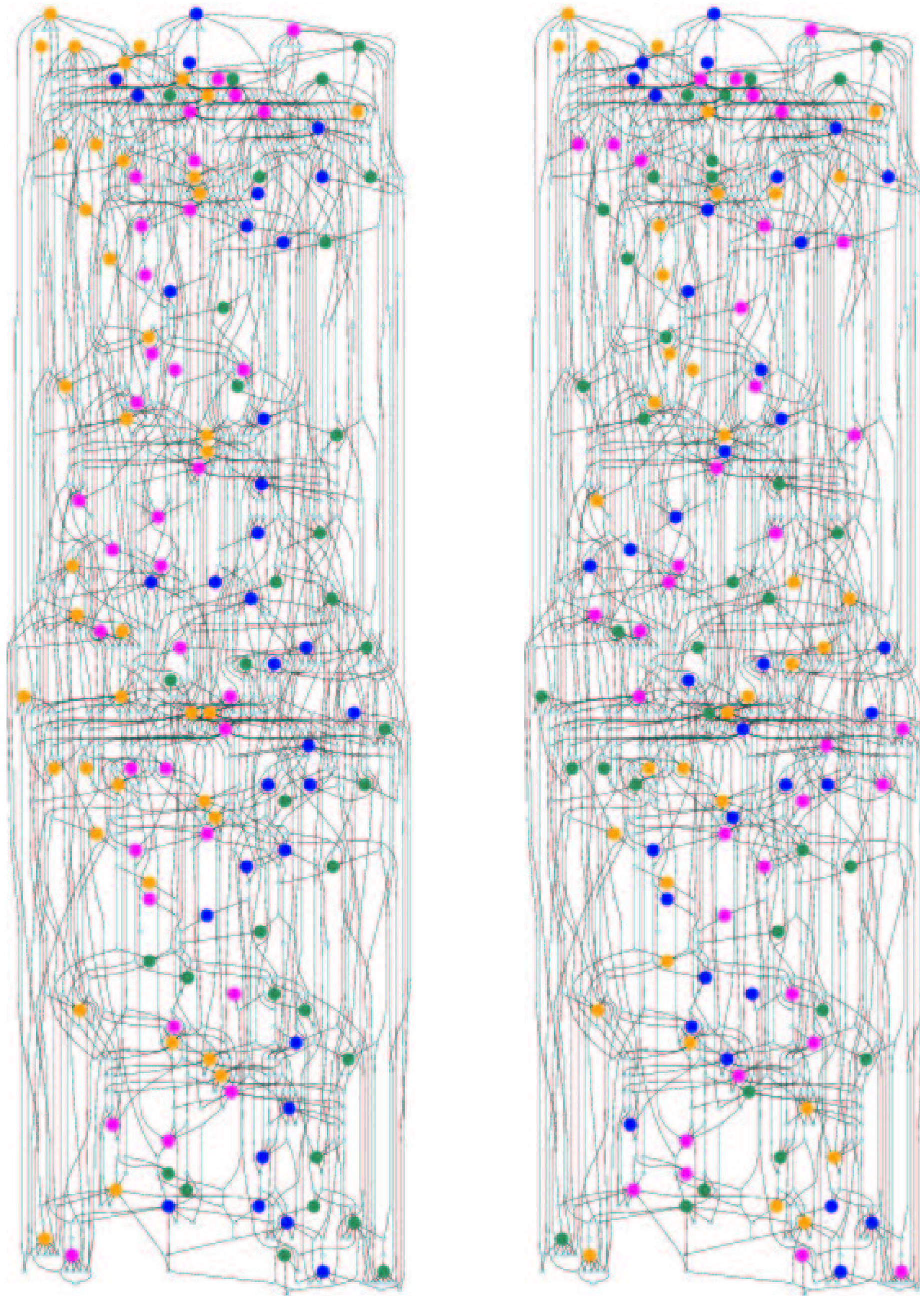


Figure 10.13 *Placement Scotch (à gauche) et Cyclic (à droite) du GFD implicite sur 2 itérations avec 1 itération du GC pour un partitionnement en 4 sous-ensembles de particules sur 4 processeurs*

Le tableau 10.12 présente ensuite les données chiffrées du nombre de tâches de calculs issues de ce GFD placées sur chacun des 4 processeurs avec la stratégie d'ordonnancement Scotch ou Cyclic.

Numéro du noeud	# de tâches exécutées (Scotch)	# de tâches exécutées (Cyclic)
0	544	571
1	271	388
2	271	403
3	164	370

TAB. 10.12 *Nombre de tâches du GFD implicite placées sur 4 processeurs lors de l'emploi des stratégies Scotch et Cyclic pour 2 itérations de la simulation partitionnée en 4 sous-ensembles de particules*

Dans le cas de la méthode implicite engendrant un grand nombre de tâches à chaque itération, le placement proposé par Scotch apparaît cette fois-ci meilleur que celui offert par l'emploi de la stratégie Cyclic. Nous allons désormais voir les répercutions de l'emploi de ces deux stratégies sur les temps des exécutions parallèles.

10.5.3 Temps de l'exécution parallèle

Les expérimentations présentées dans cette partie ont été réalisées sur deux grappes différentes. La première, appelée *I-Cluster* (projet HP, INRIA, ID-IMAG), est composée de 226 mono-processeurs Pentium III à 733 MHz avec 256 Mo de mémoire RAM, inter-connectés via un réseau de 100 Mbit/s (cf. chapitre 6). La seconde grappe, appelée *ARV*, est composée de 10 noeuds SMPs bi-processeurs Pentium III cadencés à 866 MHz avec 512 Mo de mémoire RAM inter-connectés via un réseau Ethernet de 100 Mbit/s.

Afin de nous rendre compte de l'influence de la stratégie d'ordonnancement sur les temps d'exécution de la simulation parallèle, nous présentons désormais les courbes de performances obtenues pour une taille de blocs fixée lors de l'emploi des deux stratégies d'ordonnancement vues précédemment (Cyclic et Scotch).

La figure 10.14 présente les résultats obtenus en employant la méthode de Leapfrog pour une itération d'une simulation de 490000 particules partitionnées en 234 blocs de taille 2100 lors de l'emploi des stratégies d'ordonnancement Cyclic et Scotch sur le *I-Cluster*. Puis la figure 10.15 présente les temps d'exécutions de la méthode explicite pour une itération d'une simulation de 1000000 de particules décomposées en 500 blocs de 2000 particules lors de l'emploi des stratégies d'ordonnancement Cyclic et Scotch sur la grappe *ARV*.

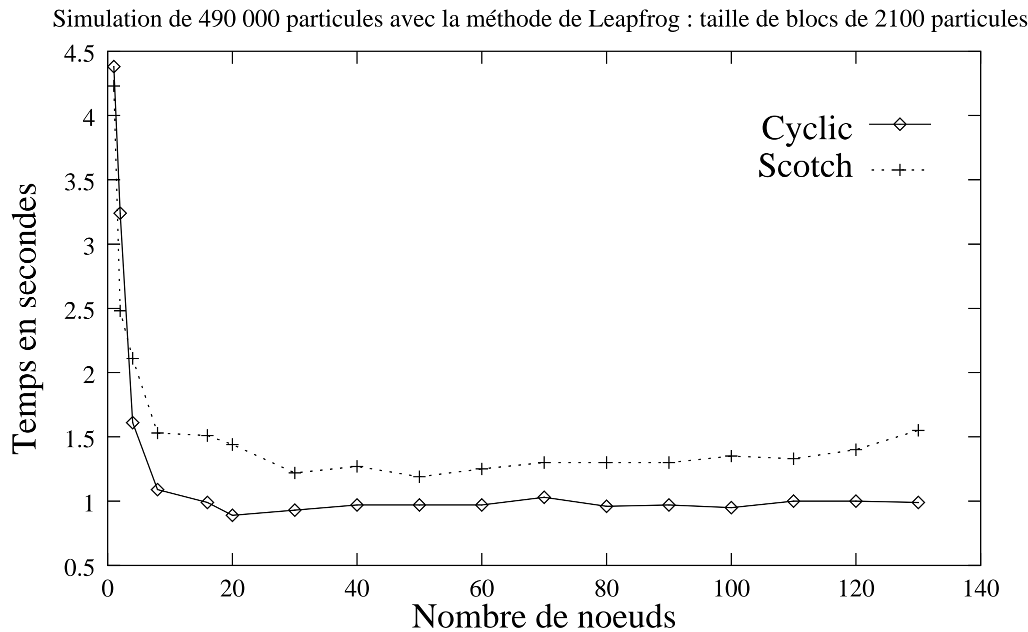


Figure 10.14 *Performances obtenues pour une itération d'un problème comportant* 490000 *particules avec une taille de blocs de 2 100 particules en employant la méthode d'intégration numérique de Stöermer-Verlet/Leapfrog sur le I-Cluster*

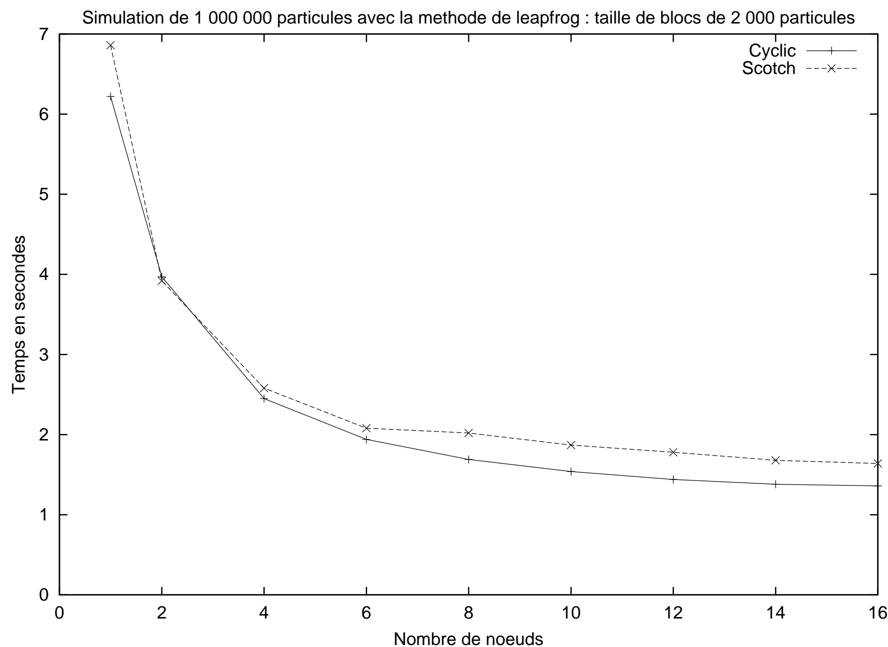


Figure 10.15 *Performances obtenues pour une itération d'un problème comportant 1 000 000 particules avec une taille de blocs de 2 000 particules en employant la méthode d'intégration numérique de Stöermer-Verlet/Leapfrog sur la grappe ARV*

Dans le cas de ces deux simulations, les courbes obtenues à la suite de l'emploi de la stratégie d'ordonnancement Cyclic et celles obtenues pour le placement proposé par Scotch ont exactement le même comportement : les temps d'exécutions sont fortement réduits jusqu'à l'emploi d'une dizaine de processeurs, pour ensuite rester stables. Par contre, lors de l'emploi de la stratégie Scotch le temps correspondant au seuil de stabilité est plus grand que celui relatif au placement Cyclic. Cela signifie que malgré un placement plus optimal offert par bibliothèque Scotch par rapport à celui obtenu en utilisant la stratégie Cyclic, son coût d'utilisation est assez important limitant l'accélération de l'application parallèle. Mais ce résultat doit être modéré par le fait que ces expérimentations portent sur des modèles de tissu totalement réguliers de 700×700 ou 1000×1000 particules. Un placement Cyclic peut alors suffire à garantir la localité des données. Par contre dans le cas de tissus irréguliers, nécessitant l'emploi d'une bibliothèque de partitionnement pour la décomposition des particules en blocs de manière à assurer la localité des données, la stratégie d'ordonnancement Scotch peut s'avérer la meilleure.

La figure 10.16 présente ensuite le cas d'une simulation de 490000 particules partitionnées en 196 blocs de 2500 particules lors de l'emploi de la méthode d'intégration de Stöermer-Verlet/Leapfrog sur le *I-Cluster*. Cette figure présente la comparaison des temps obtenus en communiquant entre processeurs la totalité des blocs ou seulement leurs frontières.

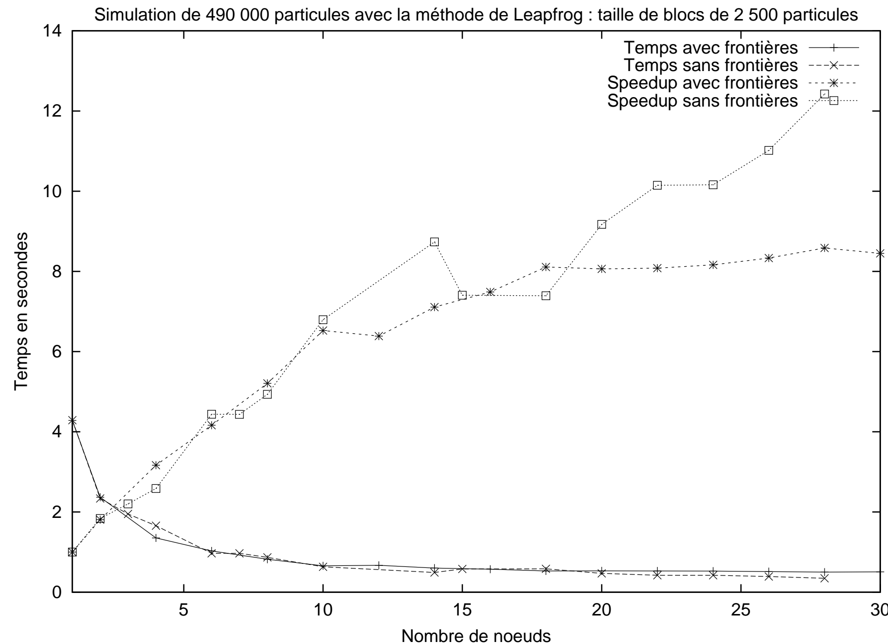


Figure 10.16 *Performances obtenues pour une itération d'un problème comportant* 490000 *particules avec une taille de blocs de 2 500 particules en employant la méthode d'intégration numérique de Stöermer-Verlet/Leapfrog sur le I-Cluster*

Dans cet exemple où les blocs comportent 2500 particules, le gain en communication obtenu en ne transférant que les frontières des blocs et non leur totalité n'est apparemment pas très significatif sur les temps de calculs. Mais il est certain qu'en pratique pour simuler de très grosses scènes, il est nécessaire de gagner en communications en effectuant des transferts moins importants. Par ailleurs, le facteur matériel limitant de la vitesse de ces communications est relatif à la latence des réseaux employés au sein des grappe de calcul : quelle que soit la taille des données envoyées, il existe un certain temps pour qu'un paquet arrive à destination. Puis en fonction de la taille des données transitées, cette latence peut devenir plus ou moins élevées. C'est pourquoi il est toujours conseillé de limiter au maximum à la fois le nombre de communications établies en assurant la localité des données, ainsi que leur volume en ne communiquant par exemple que les frontières.

Enfin la figure 10.17 présente les temps parallèles obtenus lors de la simulation de 490000 particules décomposées en 196 blocs de 2500 particules dans le cadre de la méthode d'Euler implicite sur le *I-Cluster*.

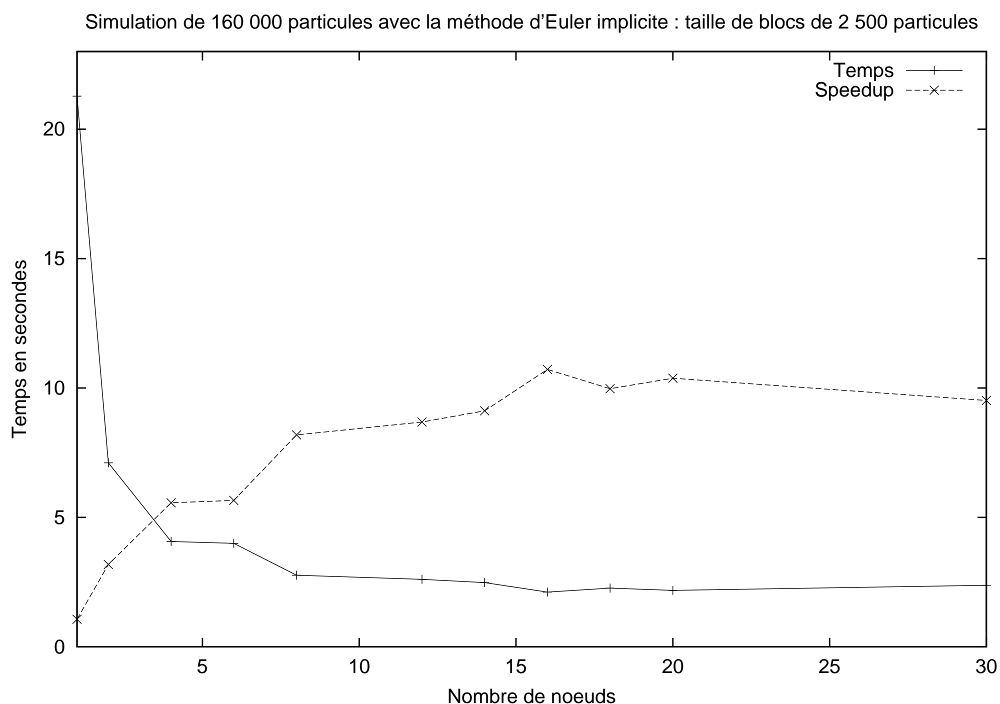


Figure 10.17 *Performances obtenues pour une itération d'un problème comportant* 490000 *particules avec une taille de blocs de 2 500 particules en employant la méthode d'intégration d'Euler implicite sur le I-Cluster*

Comme pour les courbes précédentes, cette courbe décroit très rapidement jusqu'à l'emploi d'une dizaine de processeurs pour ensuite rester stable, signifiant que le temps atteint ne peut actuellement être diminué reflétant le surcoût de la machine ATHAPASCAN.

10.6 Conclusion

Afin de paralléliser notre application de simulation de textiles, nous avons choisi d'employer l'environnement de programmation parallèle de haut niveau ATHAPASCAN. A l'exécution, le graphe de flots de données relatif au programme ATHAPASCAN est construit permettant de respecter les contraintes de précédence qui existent entre les tâches partageant des données. Ensuite des algorithmes d'ordonnancement sont utilisés avec comme objectifs un placement efficace des données et des tâches sur les processeurs.

Au sein de ce chapitre, nous avons vu tout d'abord que la simulation de textiles engendre un grand nombre de tâches de calcul ayant des temps d'exécution relativement faibles. Ce nombre de tâches est proportionnel à la fois au nombre de blocs issus du partitionnement et au nombre d'interactions qui existent entre ces blocs.

Par ailleurs le coût en mémoire est assez conséquent puisqu'il tient compte du nombre d'interactions qui existent entre particules soit environ 3 fois la taille du problème initial. Mais certaines optimisations pourraient être facilement intégrées afin de diminuer significativement la place mémoire occupée.

Ensuite nous avons mis en évidence le fait que la découpe de la simulation en sous-ensembles de particules ne doit pas être effectuée sans une étude préalable. En effet, un trop grand nombre de blocs engendre très rapidement un très grand nombre de tâches devant être traitées par ATHAPASCAN augmentant ainsi le surcoût dû à l'emploi de cet environnement.

Puis nous avons analysé deux stratégies d'ordonnancement en observant leurs placements du graphe de flots de données associé à l'application ainsi que les temps d'exécutions parallèles. Les tâches sont alors plus ou moins bien réparties sur les processeurs.

Mais au delà de ces courbes, il faut savoir que cette application de textiles a permis de mettre en évidence les points faibles de l'environnement ATHAPASCAN et ainsi de le faire évoluer tout au long de ce travail de thèse. Il faut savoir que la conception de cet environnement parallèle n'était pas forcément adaptée à un problème de type itératif à grains fins. En effet ces deux caractéristiques de l'application de textiles posent un certain nombre de problèmes. D'une part les temps de calcul de chacune des tâches étant relativement petits, les communications deviennent significatives (ce qui n'est pas le cas dans la parallélisation d'application à gros grains). Il faut alors faire en sorte que ces tâches de communications, permettant le transfert des données d'un noeud à un autre, soient prioritaires par rapport aux tâches de calculs. D'autre part, il faut noter que dans l'implantation actuelle d'ATHAPASCAN chaque processeur possède une pile contenant l'ensemble des tâches prêtes. Les processus légers peuvent ensuite aller voler au sein de cette pile des tâches afin de les exécuter. Ce vol ne devient possible qu'à la fin de chaque itération lorsque le graphe de flots de données a été généré et réparti sur les noeuds. Or ce nombre de tâches est assez conséquent : il faudrait faire en sorte que ce vol devienne

possible beaucoup plus rapidement car les threads peuvent alors être en état d'inactivité alors que des tâches doivent être exécutées. Dans le cas de problèmes récursifs, cette pile ne pose pas ce type de problème car chaque tâche engendre un grand nombre de nouvelles tâches.

Par ailleurs, il faut noter qu'il est souvent difficile d'analyser finement l'exécution d'un programme parallèle sans les outils adaptés. En effet, dans ce chapitre nous avons par exemple détaillé que des graphes de flots de données relatifs à un petit nombre de blocs devenant très rapidement illisibles. Par ailleurs les phases d'activité ou d'inactivité des processus légers ne peuvent être observées que grâce à l'emploi de traces et d'outils de visualisation de ces traces. Sans ces outils il est difficile d'observer réellement le comportement d'une exécution en distribué.

Couplage de programmes parallèles 11

En synthèse d'image, les animations 3D deviennent de plus en plus complexes nécessitant des calculs de plus en plus importants. La parallélisation des algorithmes de simulation permet une diminution du temps de calcul nécessaire à l'obtention d'une image de la scène 3D. Le calcul du rendu de la scène peut également devenir conséquent. Il devient alors nécessaire de paralléliser également la visualisation de l'animation. Mais comment faire le lien entre ces deux programmes parallèles ? Dans ce chapitre, nous montrons comment nous effectuons le couplage entre la parallélisation de la simulation de textiles exécutée sur une grappe de machines et sa visualisation sur plusieurs écrans.

11.1 Introduction

Depuis quelques années l'apparition de machines puissantes à un faible coût et les avancées en matière de réseau haut débit ont permis l'émergence de l'utilisation de grappes de machines afin d'effectuer du calcul haute performance. Il n'était dès lors plus nécessaire d'avoir recours à des machines multiprocesseurs de type Cray pour effectuer des simulations numériques complexes nécessitant de gros volumes de données. De plus l'apparition de langages de programmation parallèle haut niveau facilite la parallélisation d'applications complexes en gérant automatiquement les communications et les synchronisations entre processeurs, voire même l'ordonnancement des tâches de calcul et des données communicables.

En informatique graphique, suite à un engouement général pour les jeux vidéos, les films d'animation ou encore la Réalité Virtuelle (RV), l'animation d'objets 3D a énormément évoluée vers des algorithmes simulant des comportements de plus en plus réalistes mais également de plus en plus complexes. Ces animations nécessitent désormais une

puissance de calcul importante aussi bien pour leur simulation que pour leur visualisation.

Dans le domaine de la Réalité Virtuelle, des prototypes à base de grappes de PCs permettent de créer des systèmes immersifs multi écrans sans avoir à utiliser des machines graphiques telles que les SGI Onyx, diminuant ainsi les coûts et augmentant la flexibilité (dynamicité de l'infrastructure logicielle et matérielle) et le passage à l'échelle (prototypage et tests préalables sur petites grappes (10 PCs) et utilisation opérationnelle sur cluster de plus grande taille (200 PCs)).

Les grappes de PCs sont donc de plus en plus utilisées, mais l'enjeu scientifique important est de réussir à coupler toutes les compétences afin d'arriver à exploiter efficacement cette puissance de calcul dans des scénarios de type Réalité Virtuelle combinant la simulation d'objets 3D (fluide, végétation, etc ..), la capture de mouvements avec habillage du mannequin virtuel en temps réel (le mouvement du tissu suivant celui du personnage), l'immersion dans un centre de Réalité Virtuelle ou dans un système multi écrans de type CAVE [31], avec des interactions possibles de l'utilisateur grâce aux interfaces haptiques. L'objectif scientifique réside en effet dans l'obtention d'une méthode d'intégration de programmes parallèles de simulation 3D et de rendu, en optimisant les performances de manière à rendre possible une animation en temps réel.

Dans le chapitre 2, nous avons vu que la technique usuelle pour effectuer le calcul de l'évolution d'un objet 3D, consiste à le discrétiser en particules connectées entre elles par des ressorts, formant ainsi un maillage géométrique soumis à des contraintes internes de voisinage. Cet objet peut ensuite être soumis à des forces extérieures (pour un vêtement il peut s'agir de la gravité ou du vent), et interagir avec d'autres objets (tissu en contact avec une table, collision) entraînant des modifications au sein même du maillage. Il est utile de noter que ce maillage peut être amené à être irrégulier lors de l'utilisation de techniques dites de raffinement de maillage, dans le but d'augmenter la précision de l'évolution d'une partie seulement de l'objet (cf. chapitre 2).

Ensuite dans les chapitres 5 et 8, nous avons vu que la simulation parallèle de cet objet peut consister en la décomposition de son espace de simulation ou en la décomposition de lui-même en objets de plus petite taille. A ces objets sont ensuite associées des tâches de calculs. Celles-ci seront ordonnancées sur les processeurs en tenant compte de leurs contraintes de précédence générées par la décomposition de l'objet. Cet ordonnancement suit des critères de performances et de régulation de charge afin d'exploiter au mieux les ressources disponibles en terme de puissance de calcul et d'accès aux données ou aux I/O.

Vient ensuite la phase de visualisation de l'objet avec le couplage de la simulation parallèle à un système d'affichage multi écrans dans l'optique d'un environnement de Réalité Virtuelle immersif. Ce chapitre est consacré à cette phase. Nous verrons notamment que la gestion des communications entre ces deux applications peut être effectuée via un protocole réseau de bas niveau telles que des primitives MPI.

L'enjeu sera ensuite de contrôler les interactions entre la simulation et la visualisation afin d'effectuer des compromis entre la fréquence d'affichage et le temps de calcul avec par exemple l'emploi d'un pas de temps adaptatif pour la simulation numérique ou encore une gestion d'une cohérence seulement locale et non globale au niveau de la visualisation.

En résumé, le but est de réussir à coupler différents programmes parallèles (simulation et rendu), s'exécutant sur une même grappe ou sur des grappes différentes de machines, avec une visualisation de la scène comportant les objets simulés sur une machine graphique spécifique (pouvant utiliser un environnement multi écrans). Il faut donc trouver des méthodes permettant un partage optimal des ressources entre les programmes parallèles. D'autre part pour obtenir des animations temps réel, il faut également gérer le fait que ces programmes parallèles peuvent avoir des vitesses d'exécution différentes (fréquence d'affichage, pas de temps de l'intégration de la simulation numérique, pas de temps du rendu).

Au sein de ce chapitre, nous allons présenter l'environnement immersif de Réalité Virtuelle appelé Net Juggler qui nous a permis d'effectuer la visualisation de la simulation de textiles sur plusieurs écrans. Nous verrons ensuite comment s'effectue le couplage entre l'exécution de la simulation parallèle sur le grappe de PCs et son affichage sur une autre grappe de taille inférieure. Puis nous conclurons sur les enjeux et les apports d'une telle combinaison.

11.2 Réalité Virtuelle

Le terme populaire de la *Réalité Virtuelle (RV)* est utilisé pour décrire un ensemble de technologies qui sont générées par les ordinateurs. Il est associé à tout ce qui va des images 3D sur écran jusqu'aux technologies futures permettant de simuler des environnements aussi réels que le monde lui-même.

11.2.1 Caractéristiques

Les systèmes de Réalité Virtuelle doivent être capables de suivre l'utilisateur, d'être interactifs, d'avoir un temps de réponse en temps réel et d'avoir un affichage 3D stéréoscopique. En effet toutes ces caractéristiques sont nécessaires pour que l'utilisateur oublie qu'il se trouve dans un monde irréel et pour qu'il réagisse de la même manière que dans le monde réel.

L'ordinateur générant l'environnement de réalité virtuelle doit réagir aux mouvements de l'utilisateur pour pouvoir ajuster la perspective basée sur la position et la direction de la tête de l'utilisateur. La position et l'orientation de sa tête, ses mains et ses autres parties de son corps sont déterminées à l'aide de dispositifs spécifiques. Afin de suivre la tête

de l'utilisateur, un capteur ou un émetteur est placé dessus par l'intermédiaire de lunettes 3D ou d'un casque, permettant de détecter ses mouvements qui sont ensuite transmis à l'ordinateur.

11.2.2 Dispositifs d'affichage

Les dispositifs d'affichage employés en RV incluent aussi bien le traditionnel écran d'ordinateur, que les lunettes de RV (Head Mounted Displays, HMDs) ou encore les écrans de projection immersifs.

Ces écrans de projection immersifs constituent une alternative intéressante aux lunettes de Réalité Virtuelle permettant de produire de la même manière des images en stéréo, tout en étant moins inconfortables et en rendant l'utilisateur moins isolé des autres personnes en permettant des collaborations de groupe. En effet il est alors possible de faire interagir plusieurs personnes en même temps dans l'environnement virtuel. Bien entendu le nombre de personnes est alors limitée par les capacités matérielles à suivre au même instant plusieurs individus dans le monde virtuel.

Il est également possible d'employer non pas un seul écran plat de projection mais plusieurs afin d'obtenir un champ de vision plus important et permettant une immersion complète pour l'utilisateur dans le monde virtuel. La figure 11.1 représente un tel système appelé CAVE.

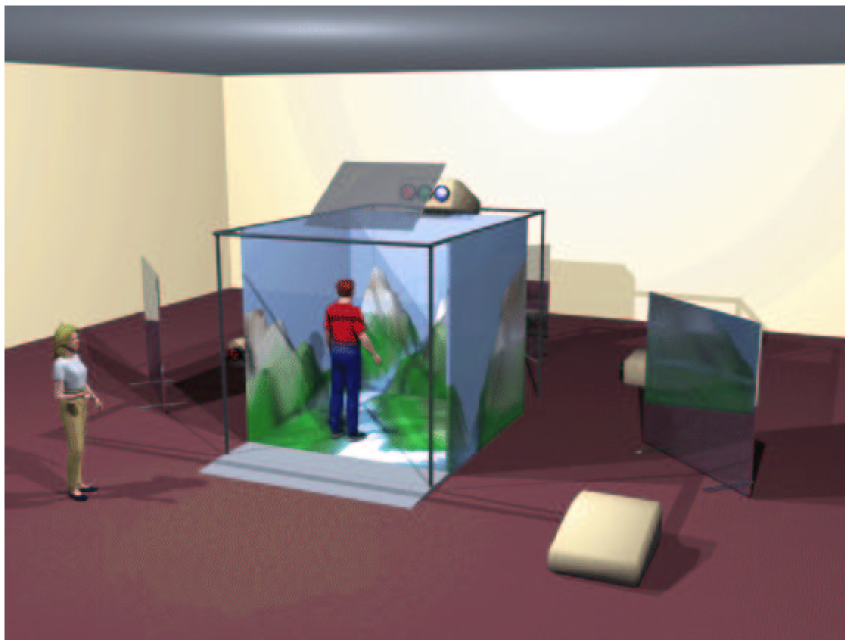


Figure 11.1 *Système immersif multiprojection de Réalité Virtuelle appelé CAVE*

11.2.3 Logiciel de Réalité Virtuelle

La Réalité Virtuelle est encore au stade du développement et il n'existe pour l'heure aucun standard de logiciel de Réalité Virtuelle. VR Juggler [68, 22] a le mérite d'être un environnement de Réalité Virtuelle assurant la portabilité des applications de façon transparente pour le développeur. De plus la configuration matérielle telle que la spécificité des écrans utilisés, peut être modifiée facilement sans aucune modification au sein de l'application grâce à une interface graphique.

11.3 Réalité Virtuelle sur cluster

Les systèmes de Réalité Virtuelle utilisent traditionnellement des stations graphiques multiprocesseurs ou des super-ordinateurs. Mais récemment l'utilisation de grappes de machines standards s'avèrent une alternative intéressante [11] offrant de nombreux avantages tels qu'un faible coût, une meilleure flexibilité et un passage à l'échelle au niveau des performances.

11.3.1 Caractéristiques

La réalité virtuelle est par nature hautement interactive et par conséquent les applications de RV ont des besoins matériels et logiciels spécifiques différents de ceux relatifs aux applications hautes performances traditionnelles. Par exemple, un système de réalité virtuelle doit pouvoir recevoir des entrées, traiter les données de l'application et produire des sorties (en général pour plusieurs écrans) au moins dix fois par seconde (voire 30 de préférence) pour conserver un niveau de performance correct.

Les logiciels courants dédiés aux clusters ne sont pas optimisés pour effectuer des communications avec des ordinateurs ou des dispositifs se trouvant en dehors du cluster. Ces communications externes doivent donc être prises en considération lors de l'élaboration d'environnements de Réalité Virtuelle sur cluster.

D'autre part l'élaboration d'applications parallèles traditionnelles s'effectue généralement en découpant les applications en tâches identiques distribuées sur l'ensemble des noeuds également de manière identique. Or la RV requiert l'exécution de tâches différentes qui sont en général de petites tailles. Comme tâches récurrentes en RV il y a les tâches réceptionnant les entrées, celles traitant l'application, ou encore celles gérant les sorties du son ou produisant les sorties vidéos. Les performances de ces tâches seraient amoindries si leurs données devaient être synchronisées sur tous les noeuds du cluster. C'est pourquoi un cluster de RV ne peut être restreint à des noeuds identiques. Des noeuds doivent donc pouvoir être spécifiques soit à la réception des entrées, ou à la génération des affichages ou encore à l'exécution d'autres tâches.

En plus d'être identiques, les noeuds employés durant la parallélisation d'une application haute performance traditionnelle, ne sont généralement pas modifiés au cours de

l'exécution. Or les applications de RV ont une forte dynamicité à laquelle le système doit savoir répondre en rajoutant, modifiant ou encore en supprimant des noeuds.

11.3.2 Environnements de RV sur cluster

Les applications de Réalité Virtuelle requièrent des fonctionnalités qui ne sont pas présentes dans les applications traditionnellement exécutées sur les clusters. C'est pourquoi différents environnements ont été conçus pour faciliter l'utilisation des grappes de machines dans ce cadre particulier de la Réalité Virtuelle.

ClusterJuggler

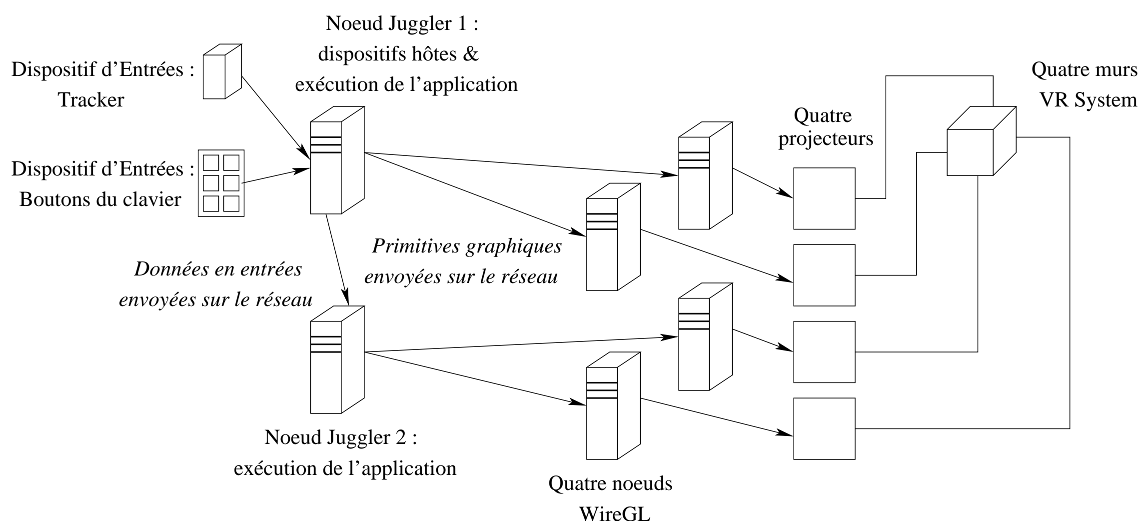


Figure 11.2 *Illustration de VR Juggler Cluster dans le cadre de la méthode hybride. L'application VR Juggler est distribuée sur deux noeuds et les primitives graphiques OpenGL sont également distribuées à l'aide de WireGL sur 4 autres noeuds du cluster.*

ClusterJuggler [90, 21] est une extension de l'architecture VR Juggler permettant l'utilisation des environnements de réalité virtuelle sur des systèmes distribués de type grappe. ClusterJuggler est constitué de plusieurs couches logicielles. La couche du dessus est responsable de la configuration de toutes les autres couches du système ainsi que des plugins. Pour cela une liste des plug-ins courants est maintenue et les appels à chaque plug-in sont synchronisés. La couche réseau permet de maintenir une abstraction de la représentation du système comprenant le cluster. Une interface de communication par messages est employée pour effectuer les communications. Durant l'exécution, il est possible de char-

ger dynamiquement des plug-ins permettant à l'utilisateur et au développeur d'ajouter facilement de nouveaux plug-ins afin de supporter de nouvelles fonctionnalités.

ClusterJuggler a implanté deux méthodes permettant l'utilisation de plusieurs écrans dans le cadre d'un environnement immersif. La première consiste à exécuter une copie de l'application sur chacun des PCs, en leur fournissant les mêmes entrées. Les copies restent synchronisées par l'intermédiaire d'un gestionnaire des entrées. La seconde consiste à utiliser la bibliothèque WireGL [65] afin d'envoyer les primitives graphiques OpenGL vers chaque PC via le réseau. Le code de l'application n'est alors exécuté que par un seul noeud du cluster, les autres se contentant d'exécuter seulement les commandes graphiques. Une combinaison de ces deux méthodes peut également être utilisée. Cette méthode hybride est illustrée par la figure 11.2.

Net Juggler

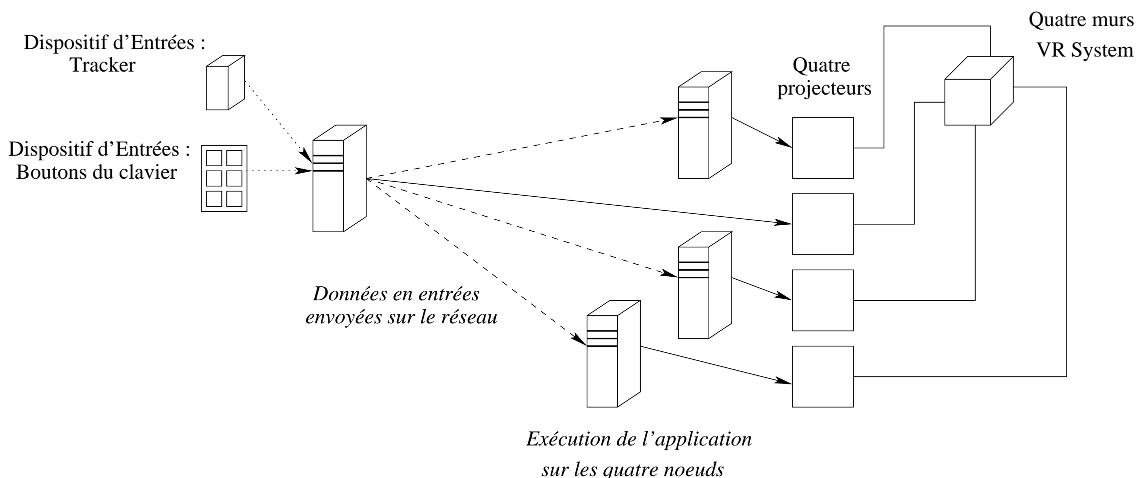


Figure 11.3 *Illustration de Net Juggler. L'application est exécutée sur chacun des noeuds. Seul un noeud reçoit les données en entrées qu'il diffuse ensuite vers tous les autres noeuds afin d'assurer la cohérence des données.*

Net Juggler [7, 5, 6, 99] est une couche logicielle se situant au dessus de VR Juggler. Il permet d'utiliser une grappe de machines, dont chacun des noeuds supporte VR Juggler, comme une unique machine VR Juggler. En d'autres termes d'un point de vue utilisateur, il n'y a aucune différence entre exécuter une application VR Juggler sur une grappe, un unique PC ou une SGI Onyx.

Une projection multi écrans de haute qualité ou un affichage stéréo nécessitent une synchronisation des différents signaux vidéo. Net Juggler n'apporte pas ce support, il est alors nécessaire d'avoir recours à du matériel approprié et/ou à des logiciels de synchronisation de signaux vidéos.

Net Juggler utilise un paradigme de parallélisation assez simple permettant d'exécuter une application VR Juggler sur une grappe. Chaque noeud de la grappe exécute sa propre copie de l'application avec ses propres paramètres. Par contre les données en entrées ne sont pas dupliquées, mais afin d'assurer une cohérence des données pour toutes les copies, les événements en entrées sont diffusés vers chacun des noeuds. Ces événements peuvent provenir des traqueurs, des gants, des claviers, ou tout autre dispositif d'entrée présents dans les applications de réalité virtuelle.

Cette parallélisation a le mérite d'être transparente pour l'utilisateur, de passer à l'échelle, et d'assurer que la quantité de données communiquée soit de faible taille. Le principal inconvénient provient de la redondance des calculs qui devrait être traité dans les versions futures de Net Juggler.

11.4 Couplage des programmes parallèles

La simulation de textiles est en fait décomposée en deux modules dont le couplage est présenté dans cette section :

1. Le premier module concerne la partie simulation à proprement parlée qui a été détaillée au sein du chapitre 9. Ses calculs ont été parallélisés pour permettre son exécution sur une grappe de PCs.
2. Le second module concerne la partie visualisation qui va être présentée au sein de ce chapitre.

11.4.1 Visualisation

La simulation de textiles est basée sur une discrétisation du tissu en un ensemble de particules. Ces particules sont connectées par des ressorts permettant d'émuler un comportement réaliste du tissu. Cet ensemble forme un maillage triangulaire. En informatique graphique, chacun de ces triangles est appelé facette. L'affichage de chacun de ces triangles va permettre la visualisation du textile comme l'illustre la figure 11.4. Cette visualisation a été réalisée en utilisant la bibliothèque graphique OpenGl [132, 112] que nous allons brièvement présenter.

Nous pouvons noter que l'objet visualisé, dans notre cas le tissu, fait partie d'un ensemble plus complexe appelé *graphe de scène*. Un graphe de scène ressemble d'une façon abstraite à un arbre. C'est-à-dire qu'il y a un certain nombre de noeuds (les objets à visualiser), qui sont connectés ensemble dans une relation transitive parent/enfant. L'idée de base de la relation parent/enfant est qu'elle exprime le contenu logique de l'enfant à l'intérieur du parent. Cela signifie, à la fois que l'enfant occupe un sous-ensemble de l'espace dans la scène que le parent occupe, et à la fois que le parent est, d'une certaine manière, responsable de son enfant. Un parent ordonnance l'agencement et l'affichage de ses enfants et peut jouer un rôle de container dans la gestion de la mémoire, la distribution

d'événements, etc ... Le graphe de scène permet ainsi de gérer plusieurs objets en même temps dans la scène.

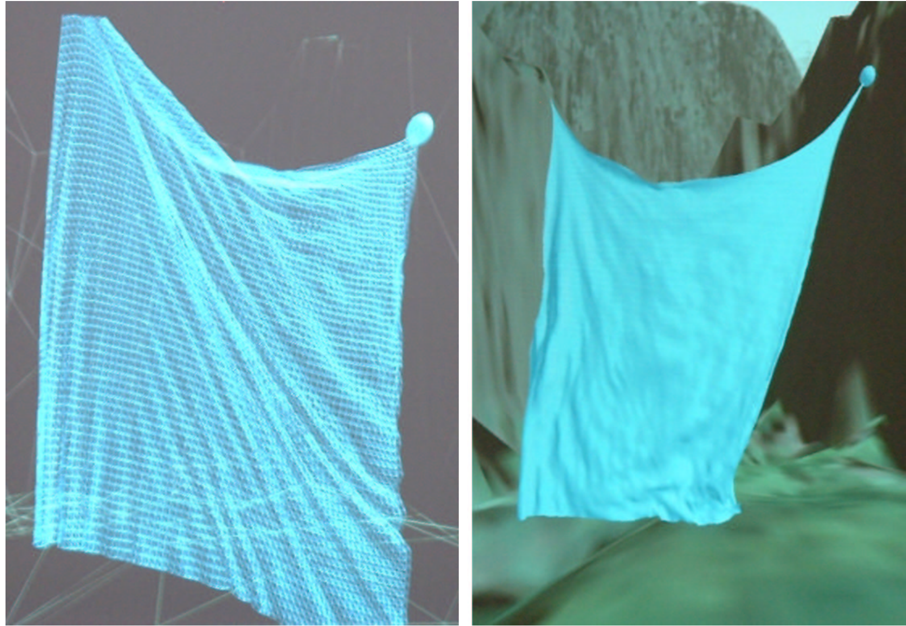


Figure 11.4 *Représentation d'un morceau de tissu. Le textile est discrétisé en un maillage triangulaire de particules. Chacun des triangles est appelé facette. Le tissu peut être affiché en mode filaire (à gauche) ou bien il est possible de lui appliqué une texture (à droite).*

Utilisation de la bibliothèque graphique 3D OpenGL

Nous reprenons pour présenter la bibliothèque OpenGL, une partie d'un document rédigé par Fabrice Neyret [88] expliquant son utilisation.

OpenGL est une bibliothèque graphique 3D. Cela signifie que le développeur lui donne des ordres de tracé de primitives graphiques directement en 3D (permettant par exemple l'affichage de sommets ou de facettes), une position de caméra, des lumières, des textures à plaquer sur les surfaces et qu'à partir de là OpenGL se charge de faire les changements de repère, la projection en perspective à l'écran, le clipping (élagage des faces qui sortent de l'écran et découpe des faces partiellement à l'écran), l'élimination des parties cachées, d'interpoler les couleurs, et enfin de rasteriser les faces c'est-à-dire de les tracer ligne à ligne pour en faire des pixels.

Pour cela OpenGL s'appuie sur les ressources matérielles disponibles selon la carte graphique. Toutes les opérations de base sont a priori accessibles sur n'importe quelle

machine équipée d'une carte graphique. La différence réside simplement dans leur temps d'exécution variant selon si elles ont été implantées à un niveau matériel ou à niveau logiciel.

Toute **primitive surfacique 3D est décomposée en triangles par OpenGL**. Le triangle, la ligne et le point sont donc les seules primitives géométriques traitées par le matériel, ce qui ramène toutes les interpolations au cas linéaire. Ceci est en effet facile à traiter de façon incrémentale au niveau matériel, expliquant cette limitation aux triangles, mais qui peut produire un aspect légèrement anguleux. C'est pourquoi les **facettes sont généralement normalisées afin de lisser la surface de l'objet affiché**.

Par conséquence, le fichier d'entrées du module de visualisation comporte la liste des numéros des sommets de chacune des facettes constituant le textile, les sommets étant regroupés trois par trois puisque dans notre cas les facettes sont des triangles. Ensuite à partir de la connaissance de ces facettes, le maillage du tissu est reconstitué entièrement permettant de connaître le voisinage de chacune des particules. Il n'y a donc aucune hypothèse de départ sur le type de maillage (régulier ou non), permettant de visualiser en pratique tout type d'objet correctement décomposé en facettes.

L'affichage des facettes du tissu s'effectue en spécifiant les coordonnées 3D de ses trois sommets, c'est-à-dire en fournissant les positions géométriques de chacune des trois particules. Pour éviter l'apparition de points anguleux à la jonction des facettes, la normale de chacun des sommets est également calculée et fourni à OpenGL lors de l'affichage des sommets.

Plusieurs repères interviennent dans la description d'une scène. Les coordonnées ne sont pas toutes données dans un unique repère monde. Ceci permet de changer facilement de point de vue sans modifier la description des objets, ou de changer la position des lumières, ou encore de modifier l'orientation des objets. Ceci permet en outre de réutiliser facilement des descriptions de parties d'objets par simple changement de repère.

Il existe **plusieurs façons de spécifier l'apparence d'une facette** (figure 11.4) :

- *En définissant la couleur de la face.* Une primitive permet de définir les couleurs RGB relatives à la face définie par ses sommets. L'opacité peut également être précisée en ajoutant un paramètre de transparence.
- *En spécifiant la couleur aux sommets à interpoler sur la face.* Cette couleur est redéfinie pour chaque sommet juste avant la primitive permettant l'affichage du sommet.
- *A l'aide du modèle d'illumination de Phong*[1] *en fonction de l'orientation par rap-*

[1] Illumination locale de Phong : formule décrivant la façon dont une surface renvoie la lumière dans les diverses directions, contribuant à caractériser l'aspect du matériau. Phong est un modèle particulier, isolant les effets de la lumière ambiante, diffuse ou spéculaire (reflets de la source), et prévoyant une couleur spécifique à chacun d'entre eux.

port aux lumières. Un tel modèle nécessite la définition d'une matière composée d'une couleur ambiante (celle qui apparaît dans l'ombre), une couleur diffuse (celle qui apparaît du côté éclairé, souvent la même), une couleur spéculaire (celle des reflets, blanche pour du plastique), et un coefficient de rugosité (qui contrôle l'épaisseur de la tache spéculaire). Toutes ces données peuvent être spécifiées en entrée du module de visualisation et peuvent même être modifiées au cours de la simulation. L'illumination tient compte de la direction de la lumière et de l'observateur par rapport à l'orientation de la facette. Il faut donc préciser cette dernière à chaque sommet avant l'affichage du sommet. Il est également possible de laisser OpenGL estimer tout seul les normales, mais c'est coûteux et moins précis. En outre il est également possible de lui demander de normaliser les normales, ce qui est notamment utile lors de la spécification de la déformation de l'objet (ne serait-ce qu'un scaling), mais cela entraîne un calcul supplémentaire par sommet, incluant l'évaluation d'une racine carrée, sachant que chaque sommet est redéfini pour toutes les faces auxquelles il appartient.
– *En plaquant une texture sur la face.* La description du motif est séparée de la spécification de son plaquage, lequel est donné en fixant les valeurs des coordonnées texture (u,v) à l'aide d'une primitive à chaque sommet avant l'affichage de ce dernier. La projection sur toute la surface est obtenue par l'interpolation des coordonnées de texture sur les facettes. La correspondance entre la position sur la “tapisserie” et la position dans le monde 3D est ainsi faite. En entrée du module de visualisation de la simulation de textiles, il est ainsi possible de fournir un fichier de coordonnées de texture permettant ce plaquage.

Dans sa version actuelle, OpenGL ne s'occupe pas de l'interface graphique 2D, donc en particulier ni de l'ouverture des fenêtres, ni de la gestion des évènements tels que des clics de souris. Ces tâches dépendant du système d'exploitation et de la machine, elles sont laissées à d'autres bibliothèques. Plusieurs bibliothèques sont en effet disponibles pour faciliter la gestion de l'interface sans descendre dans les bas-fonds du serveur X. GLUT [73], fournie avec OpenGL, est simple d'utilisation et très portable. Pour l'utiliser, seulement quelques lignes sont à rajouter dans le programme principal. Pour notre part, nous avons opté pour l'environnement de Réalité Virtuelle multi-projecteurs Net Juggler, présenté dans la section 11.3.2, basé sur la bibliothèque VR Juggler.

Utilisation de l'environnement Net Juggler

L'utilisation de l'environnement Net Juggler est assez simple, puisque comme les autres bibliothèques basées sur OpenGL il suffit de rajouter que quelques lignes spécifiques à cet environnement au sein de notre programme. Mais son emploi permet d'obtenir une application de réalité virtuelle multi-projecteurs basée non pas sur l'emploi de calculateurs puissants, mais sur une grappe de machines standards [99]. La figure 11.5

présente la visualisation de la simulation de textiles sur trois écrans.

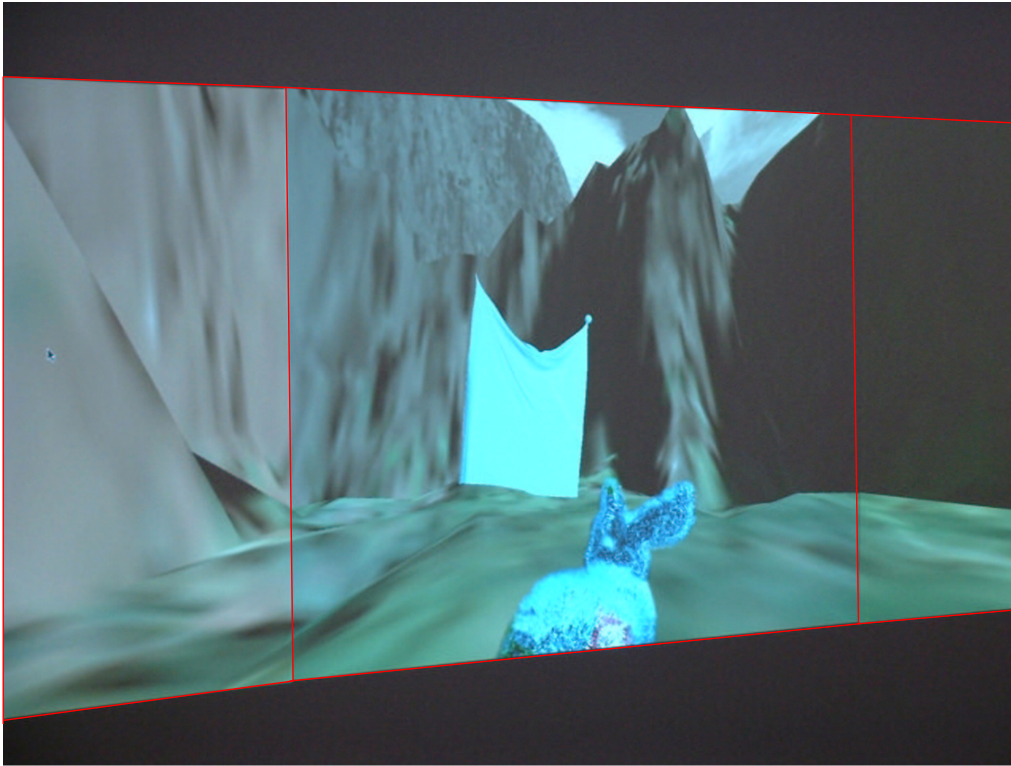


Figure 11.5 *Visualisation de la simulation de textiles sur trois écrans via l'environnement de réalité virtuelle multi-projecteurs Net Juggler*

Le module de visualisation est dupliqué par Net Juggler sur tous les noeuds de la grappe servant uniquement à cette effet. L'environnement intercepte et distribue également les évènements de contrôle tels que les clics souris, etc ... La configuration de la grappe s'effectue à l'aide de fichiers de configuration. Ils permettent par exemple de spécifier la vue de la caméra relative aux différents écrans, d'expliciter lors de l'utilisation d'un traqueur le noeud qui en bénéficie, etc ... Ensuite Net Juggler se charge de la configuration de chacun des noeuds. Du point de vue de l'utilisateur, mise à part cette phase de configuration, l'utilisation d'une seule machine ou d'une grappe de PCs est effectuée de manière totalement transparente.

11.4.2 Couplage

Après avoir détaillé la partie simulation parallèle dans le chapitre 9 et le module de visualisation dans la section précédente, nous allons désormais nous intéresser au couplage de ces deux modules, qui fut l'objet de deux articles référencés en [8] et [136]. Ce

couplage entre une simulation parallèle et une visualisation parallèle peut notamment permettre une meilleure compréhension de phénomènes complexes dans des domaines tels que celui de la physique ou encore celui de la chimie.

Pour récapituler nous devons coupler la simulation de textiles exécutée sur une grappe de machines avec son module de visualisation, lui-même distribué sur une grappe de machines. Ces deux parties peuvent potentiellement être exécutées sur deux grappes distinctes ou au contraire sur la même. Il faut donc réussir à établir une communication entre ces deux modules. Afin que ces communications soient les plus petites possibles, seules les positions des particules calculées par la simulation doivent être transmises au module de visualisation au cours de l'exécution.

Liaison entre les deux modules

Pour effectuer ce couplage, une simple liaison TCP a été établie entre le programme de simulation parallèle et celui de visualisation. La figure 11.6 illustre ce couplage dans le cas où deux grappes de machines sont employées, la première servant pour effectuer les calculs de la simulation parallèle (ATHAPASCAN), tandis que la seconde sert uniquement à la visualisation multi écrans (Net Juggler).

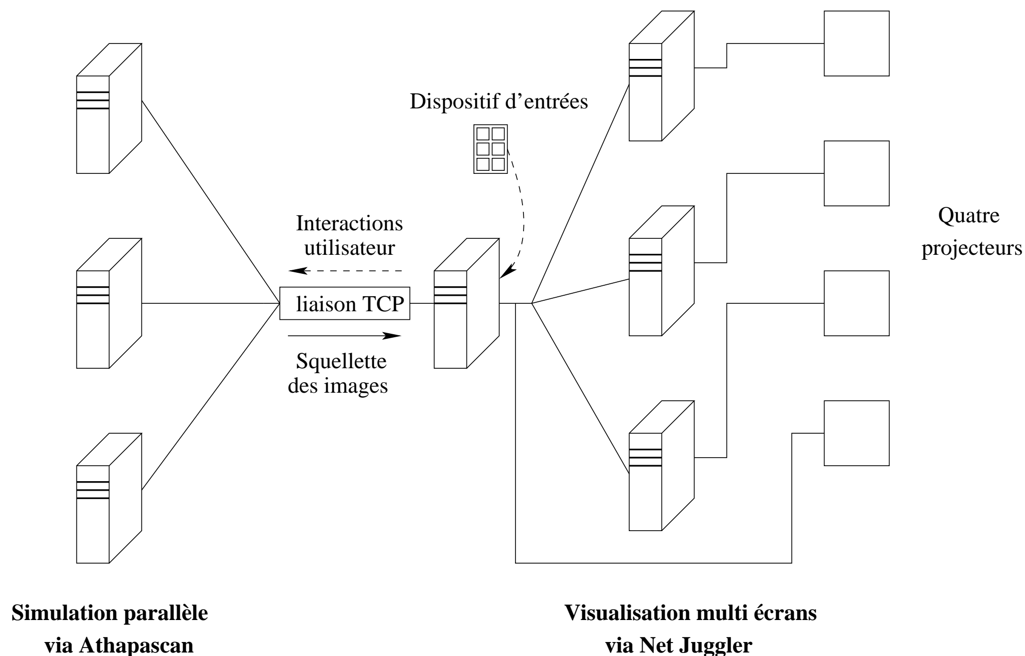


Figure 11.6 *Couplage de la simulation parallèle (*ATHAPASCAN*) avec une visualisation multi écrans (Net Juggler). Les deux parties de l'application s'exécutent sur des grappes de machines différentes.*

La figure 11.7 illustre le cas où une seule grappe est employée à la fois pour exécuter la simulation parallèle et pour effectuer le rendu de l'application. Le nombre de noeuds servant à la visualisation étant limité par le nombre d'écrans, certains des noeuds de la grappe servent à la fois à la simulation et à la visualisation, tandis que d'autres exécutent uniquement des tâches ATHAPASCAN.

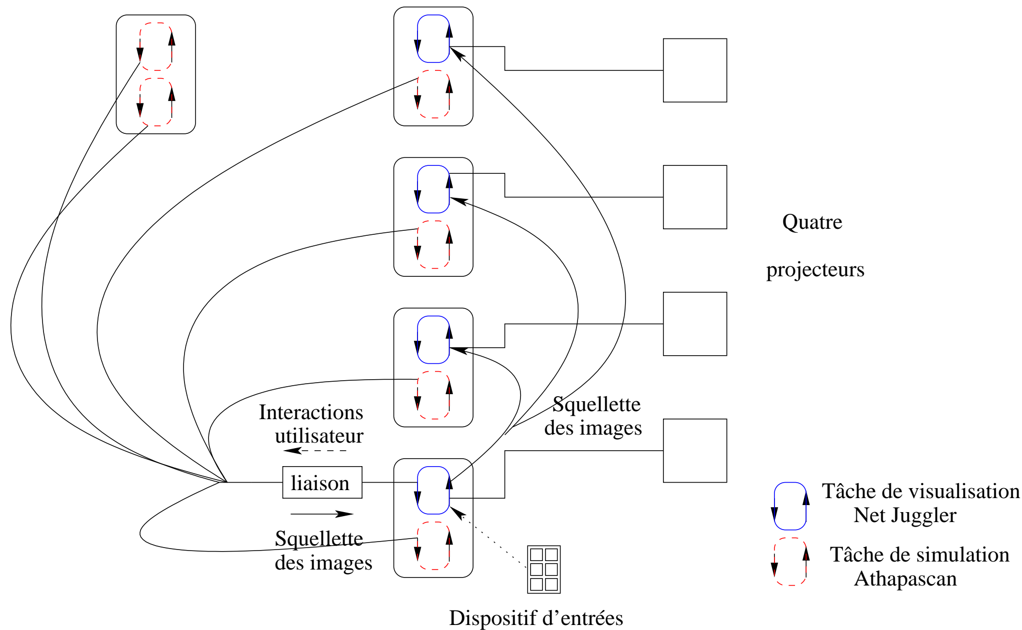


Figure 11.7 *Couplage de la simulation parallèle (*ATHAPASCAN*) avec une visualisation multi écrans (Net Juggler). Les deux parties de l'application s'exécutent sur la même grappe de machines.*

Un dialogue de type client/serveur s'instaure alors entre les deux modules afin de faire transiter les positions des particules calculées à chaque pas de temps de la simulation vers le module de visualisation. Dans la version actuelle de l'implantation de la simulation de textiles, la liaison n'est effectuée qu'avec une seule machine du coté de la visualisation. Puis ce noeud se charge de diffuser les données reçues à toutes les autres machines de la grappe.

Modèle asynchrone

Pour obtenir des animations en temps réel, il faut gérer le fait que ces programmes parallèles ont des vitesses d'exécution différentes : la fréquence d'affichage, le pas de temps de la simulation, le pas de temps de l'intégration de la simulation numérique (ex : le nombre d'itérations de la méthode du Gradient Conjuguée utilisée dans le cadre de la méthode d'Euler implicite), le pas de temps du rendu, etc ... Dans la version actuelle de

notre couplage, l'envoi des positions de la simulation vers la visualisation ne s'effectue que pour les pas de temps correspondants à l'affichage d'une image. Sachant qu'il faut afficher 25 images par seconde dans le cadre d'une animation en temps réel, cet envoi ne se réalise que pour les pas de temps multiples de $1/25 = 0.04s$. Cela permet d'éviter de saturer inutilement la liaison TCP.

Il est important de souligner qu'aucune synchronisation n'est réalisée entre le module de simulation et celui de visualisation. C'est-à-dire que le programme de simulation n'attend pas que la visualisation ait finie d'afficher une image pour continuer à effectuer ses calculs, et réciproquement le module de visualisation n'attend pas la réception de toutes les données d'un pas de temps donné pour commencer l'affichage de l'image correspondante. Le modèle construit est ainsi totalement asynchrone.

Interaction avec l'utilisateur

L'environnement Net Juggler permet de capturer facilement des évènements extérieurs tels qu'un clic de souris, la position de la souris, des évènements claviers, etc ... Dans le cas où le souhait est de permettre à l'utilisateur de modifier la position d'une particule donnée du tissu en bougeant la souris, il suffit de récupérer la position de celle-ci et de la renvoyer au programme de simulation qui en prendra compte lors des calculs des pas de temps suivants. La figure 11.8 montre la modification "à la main" de la position d'une particule du tissu.

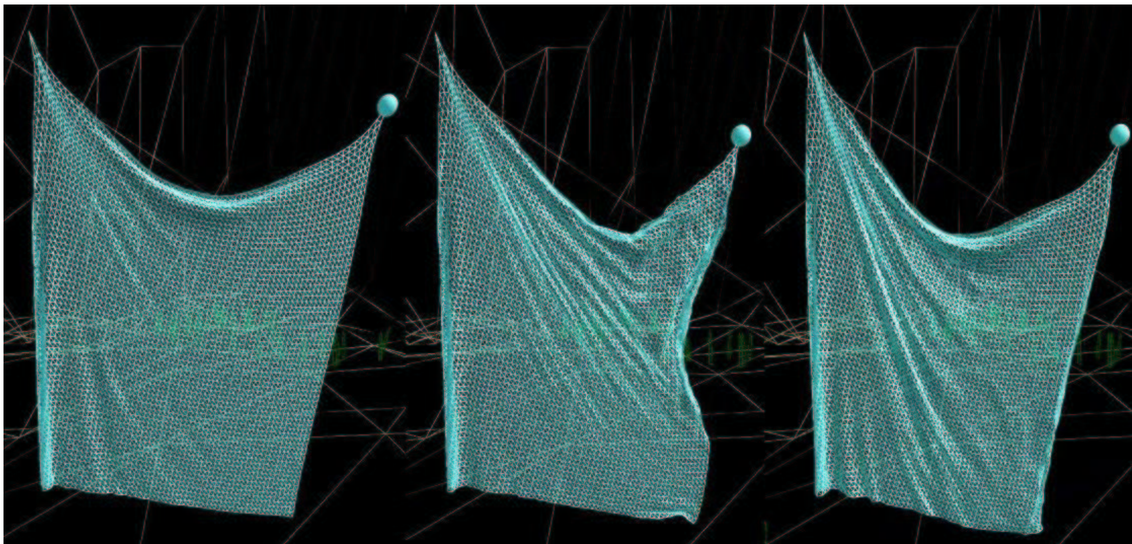


Figure 11.8 *Interaction de la simulation de textiles avec l'utilisateur. Le coin en haut à gauche est fixé, tandis que le coin en haut à droite est dirigé par l'utilisateur via la souris.*

Données transférées

Dans un souci de diminuer au maximum les communications, seules les positions sont communiquées entre le programme de simulation et celui de la visualisation. Si la simulation comporte n particules, il y a $3 \times n$ données à communiquer soit $4 \times 3 \times n$ octets si les données sont des flottants ou $8 \times 3 \times n$ pour des doubles.

Ainsi pour 100 000 particules codées en flottants, les données transférées pour chaque itération représentent un volume de 1.2 Mo. Comme nous utilisons un réseau ayant un débit d'un Gigabit par seconde soit 125 Mo par seconde, le temps réel au niveau du transfert est assuré car il est alors possible de transférer 5 Mo toutes les 0.04 secondes.

Mais si la simulation implique un million de particules, le transfert pour une itération représentant alors un volume de données de 12 Mo, une seule connexion assurant le transfert de 5 Mo est alors insuffisante pour du temps réel. Elle représente même un goulot d'étranglement pour l'application : il devient alors nécessaire d'effectuer plusieurs connexions entre les deux parties de l'application.

11.4.3 Améliorations

L'élaboration du couplage entre différents programmes parallèles représente un vaste domaine de recherche. Suite à notre expérience sur la simulation de textiles, nous pouvons déjà mettre en avant plusieurs axes de recherche :

Gestion du flux d'entrées/sorties

L'enjeu est de contrôler les interactions entre la simulation et la visualisation. Il y a notamment des compromis à effectuer entre la fréquence d'affichage et le pas de temps de calcul avec par exemple l'emploi d'un pas de temps adaptatif pour la simulation numérique ou encore la gestion d'une cohérence seulement locale et non globale au niveau de la visualisation. Il est également possible d'imaginer que dans le cadre de grosse scène la visualisation soit plus lente que la simulation. Il serait alors intéressant d'augmenter la précision des calculs diminuant ainsi leur vitesse d'exécution.

Compression des données

Dans les cas où les communications devant transiter entre le module de simulation et celui de visualisation sont trop importantes, il peut toujours être intéressant de compresser les données.

Type de liaison

La liaison établie actuellement entre les deux modules de simulation et de visualisation est de type N-1. Il n'y a en effet qu'une seule machine de la grappe de visualisation

qui reçoit les positions calculées par la simulation. Cette machine représente donc potentiellement un goulot d'étranglement. Il serait alors préférable de faire en sorte que toutes les machines de la grappe de visualisation reçoivent les données calculées créant ainsi une liaison de type N-N. Ce type de liaison est illustrée par la figure 11.9.

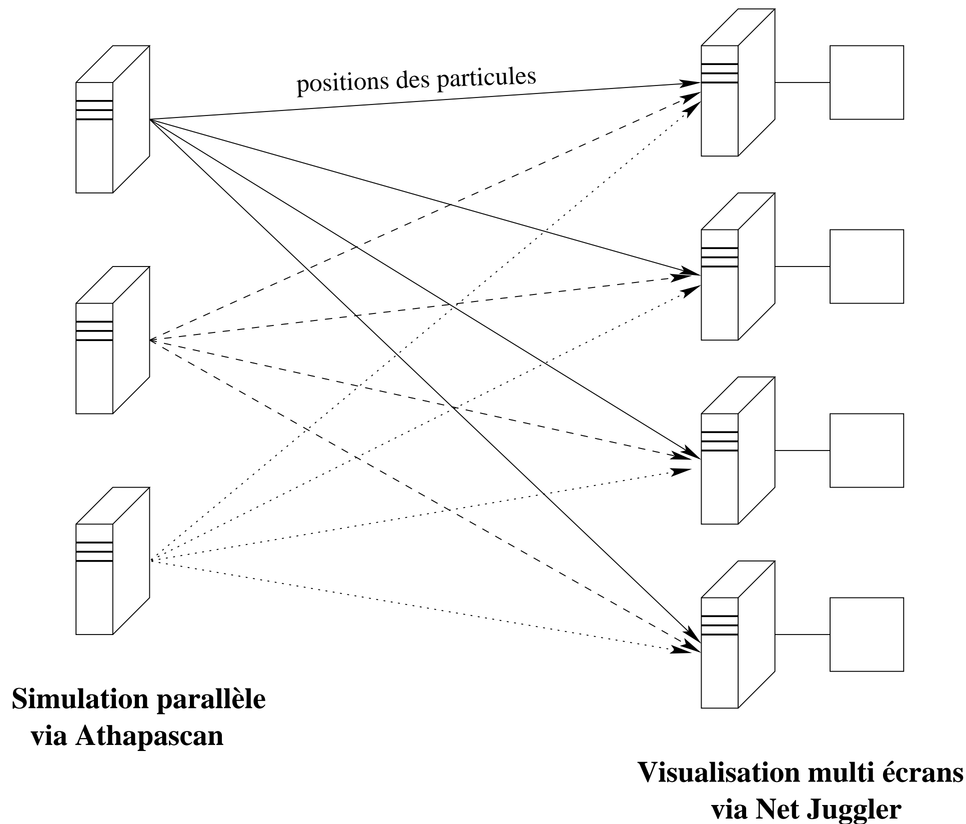


Figure 11.9 *Liaison TCP de type N-N entre la grappe de simulation et celle de visualisation*

Mais hélas ce type de liaison augmente considérablement le nombre de communications effectuées sur le réseau. D'autre part ce type de couplage passe difficilement à l'échelle à la différence de la partie simulation parallèle. C'est pourquoi il serait intéressant de trouver une solution qui passe également à l'échelle au niveau du couplage.

11.5 Conclusion

Dans le monde de la Réalité Virtuelle, les grappes de machines ont également fait leur apparition. Elles permettent notamment l'obtention d'environnements immersifs de type multi-projection à un moindre coût, c'est-à-dire sans avoir à recourir à l'utilisation de puissantes machines graphiques telles que les SGI Onyx. Par contre les caractéristiques

bien particulières des applications de réalité virtuelle, telles que la possibilité de faire interagir l'utilisateur en temps réel dans le monde virtuel, ont rendu inutilisables l'emploi direct d'environnements parallèles déjà existants.

Dans le cadre de notre simulation de textiles, nous avons choisi d'utiliser l'environnement de réalité virtuelle immersif Net Juggler. La visualisation est ainsi également réalisée sur une grappe de machines.

A l'heure actuelle, le couplage des deux programmes parallèles, la simulation et la visualisation, repose sur une liaison TCP. La simulation envoie régulièrement les positions des particules qu'elle a calculées au module de visualisation qui se charge de les afficher. D'autre part, l'utilisateur a la possibilité de faire bouger le tissu à l'aide de la souris en modifiant par exemple la position d'une particule donnée. Cette interaction est capturée par la visualisation qui informe ensuite la simulation afin qu'elle prenne en compte cette nouvelle position. Les communications s'effectuent ainsi dans les deux sens.

A terme, le couplage de programmes parallèles peut s'avérer un élément déterminant permettant la compréhension de phénomènes complexes nécessitant à la fois d'énormes ressources de calcul et d'affichage.

Conclusion et perspectives 12

Dans cette thèse, nous nous sommes intéressés à la parallélisation d'une application de simulation physique de textiles dans le domaine de la synthèse d'images. Les points suivants ont été abordés :

– Nous avons mis en évidence les **caractéristiques** relatives à une **simulation physique d'objets déformables**. Les objets sont discrétisés en un ensemble de particules. Les déformations subies par l'objet sont ensuite reproduites grâce à la présence de ressorts entre les particules créant des forces d'interaction. La simulation consiste alors à calculer la position de ces particules en intégrant leur accélérations fournies par l'application de la loi fondamentale de la dynamique.

– Nous avons résumé les **principales étapes** de la **conception d'un algorithme parallèle** dont la **parallélisation** est basée sur un **partitionnement du problème initial**. Nous avons également mis en évidence les avantages liés à l'emploi d'une machine parallèle de type grappe par rapport à une machine multiprocesseur traditionnelle, ainsi que l'utilisation d'environnements de programmation parallèles de haut niveau plutôt que ceux basés sur des échanges de messages.

– Nous avons proposé une stratégie de **partitionnement** basée sur un **découpage de l'objet** permettant par la suite une répartition quasi uniforme de la charge de calculs sur les processeurs.

– Nous avons présenté la **parallélisation implantée** à l'aide de l'**environnement** ATHAPASCAN. Celle-ci est basée sur l'exécution de tâches asynchrones, les dé-

pendances de données entre ces tâches étant gérées via la construction du **graphe de flots de données**. Celui-ci est connu par l'ensemble des processsus légers exécutant le programme ATHAPASCAN.

- Cette implantation a été testée sur des **grappes machines**. Différents algorithmes d'**ordonnancement** ont été utilisés en analysant pour chacun d'entre eux la répartition des tâches de calculs sur les noeuds de la grappe. Nous avons également mis en évidence l'importance du choix de la **granularité** de la parallélisation, qui n'est autre dans notre cas que la taille des blocs de particules issus du partitionnement. Par ailleurs trois **méthodes d'intégration** (Euler explicite, leapfrog et Euler implicite) ont été intégrées dans cette simulation parallèle.

- Nous avons élaboré l'**affichage de la simulation** à l'aide d'un **couplage** permettant le transit des données calculées de la partie calcul parallèle vers la partie visualisation. Par ailleurs, la **visualisation** a été réalisée en utilisant l'**environnement Net Juggler** permettant de tirer profit de la puissance de plusieurs cartes graphiques présentes sur une grappe de machines. Cet environnement facilite également l'ajout d'un **caractère interactif** à la simulation en permettant à l'utilisateur de faire "bouger" le tissu simulé.

- Le travail mené dans cette thèse a imposé l'ajout d'un contrôle de l'ordonnanceur au sein d'ATHAPASCAN afin d'autoriser dans les applications parallélisées de type simulations l'emploi d'une boucle infinie.

Cette thèse a abouti à l'obtention d'une plate-forme permettant d'effectuer une simulation de textiles en parallèle qui peut être visualisée sur plusieurs écrans. Cette plate-forme issue d'un travail de recherche nécessite encore un grand nombre d'améliorations. L'une d'entre-elles consiste en la complexification du modèle physique employé par l'ajout par exemple de ressorts de flexions. Cet ajout permettrait d'affiner très facilement la courbure des tissus.

Par ailleurs, il faudrait rajouter à notre simulation de textiles les phases de détection et de traitement des collisions du tissu avec lui-même ou avec des objets extérieurs. Pour la première phase de détection il est nécessaire de prendre en compte les aspects géométriques des objets contenus dans la scène. Les algorithmes proposés à l'heure actuelle sont généralement basés sur des hiérarchies de boîtes englobantes [82, 53, 74]. Dans notre situation, il est facile d'adjoindre à chaque bloc de particules une boîte englobante. Une structure hiérarchique des boîtes serait ensuite superposée. Le point intéressant qui nous semble prometteur est que la hiérarchie des boîtes pourrait être couplée à l'ordonnanceur dynamique. Ainsi la gestion des collisions pourraient être améliorée par le fait que les boîtes "voisines" dans l'espace de l'objet géométrique seraient traitées sur des processeurs "voisins".

Puis la seconde phase concernant le traitement des collisions s'effectue par l'ajout de nouveaux ressorts entre les particules. Cet ajout va modifier quelque peu les matrices des contributions des forces. Mais ces modifications n'affectent en rien les propriétés de ces matrices qui sont symétriques et creuses. Elles ne rajoutent en effet qu'un nombre limité d'interactions locales aux particules. Ces changements vont alors engendrer des modifications du graphe de flot de données associé à l'application. Mais ce dernier étant régénéré à chaque itération de la simulation, ces changements seront relativement transparents.

Il resterait alors à valider cette hypothèse remettant en avant le compromis entre le découpage de l'espace de simulation et celui de l'espace objet. La mise au point dépendra alors du "degré de collisions" observé dans la scène par rapport à la taille des objets manipulés.

Ce projet de recherche se poursuit actuellement dans le cadre du pré-projet MOAIS (INRIA, ID-IMAG) par la volonté de mettre au point des environnements immersifs avec des contraintes de temps réel mou sur la gestion des entrées/sorties. Plusieurs problèmes se posent alors : d'une part il faut assurer le temps réel au niveau de la visualisation de l'application avec la génération de 24 images par seconde, et d'autre part ce temps réel doit également être assuré par les différents dispositifs de Réalité virtuelle comme par exemple par les capteurs de mouvements générant des entrées pou la simulation, ou encore des outils haptiques comme un bras à retour d'effort constituant cette fois ci une sortie de la simulation. Afin d'assurer ce temps réel au niveau de la simulation, il serait intéressant de mettre en place des outils de contrôle de la simulation obéissant à des contraintes. Une des questions qui se posent alors est : comment pourrions-nous intégrer ces contraintes au sein de l'ordonnanceur afin qu'il puisse gérer au mieux le niveau de qualité des calculs parallèles ?

IV

Annexes

Méthode du Gradient Conjugué A

La difficulté majeure dans l'implantation d'une méthode implicite réside dans la résolution d'un système linéaire creux. En effet un tel système requière des structures de données appropriées permettant la gestion de matrices creuses de grande taille et permettant également d'effectuer les opérations nécessaires sur ces matrices. L'emploi de la Méthode du Gradient Conjugué va permettre d'éviter ces difficultés.

A.1 Introduction

Le mouvement dynamique d'un textile est reproduit à l'aide d'un système à particules connectées par des ressorts. La position de chacune des particules, et les liaisons entre elles permettent de reproduire le comportement du tissu au cours du temps. Il suffit pour cela d'identifier toutes les forces exercées sur les particules, d'appliquer ensuite la loi fondamentale de la dynamique et d'intégrer les accélérations ainsi obtenues pour connaître la position de ces masses ponctuelles. Il existe essentiellement deux types de méthode d'intégration : les méthodes explicites et les méthodes implicites. Les méthodes implicites conduisent à la résolution d'un système non-linéaire. La méthode d'intégration d'Euler implicite par exemple, réduit le problème d'intégration à la résolution du système non-linéaire

$$Y_1 - hf(Y_1) - Y_0 = 0. \tag{A.1}$$

Dans le cadre de la simulation de textile, Baraff et Witkin [13] ont montré que ce système non-linéaire peut être transformé en un système linéaire. En effet l'application du schéma d'intégration d'Euler implicite sur le système à particules conduit à la résolution du système

$$\left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)\Delta v = h\left(f(t) + h\frac{\partial f}{\partial x}v(t)\right)$$

permettant de trouver les vitesses des particules. Nous pouvons observer que ce système linéaire est de la forme $Ax = b$ avec A une matrice symétrique définie-positive, si nous

définissons la matrice A, le vecteur solution x et le vecteur b par

$$A = \left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right), \quad x = \Delta v \quad b = \left(hf(t) + h^2\frac{\partial f}{\partial x}v(t)\right). \tag{A.2}$$

Il existe une multitude de méthodes permettant la résolution d'un tel système. Ces méthodes sont généralement répertoriées en deux catégories à savoir les méthodes *directes* et les méthodes *itératives*. Les méthodes directes trouvent la solution exacte en un nombre fini d'opérations généralement d'ordre N^3, si N est la taille des vecteurs b et x et A est une matrice de taille $N \times N$. Les méthodes itératives, quant à elles, ne permettent pas d'obtenir la solution exacte du système $Ax = b$ en un temps fini, mais elles convergent asymptotiquement vers une solution. Néanmoins les méthodes itératives génèrent souvent une solution ayant une précision correcte, après un nombre relativement petit d'itérations et sont alors préférables aux méthodes directes. Ceci est généralement le cas quand N est important. De plus les méthodes itératives nécessitent moins de mémoire que les méthodes directes dans le cas où A est une matrice creuse.

Ce chapitre présente la méthode du Gradient Conjugué [15] qui est une méthode itérative particulièrement bien adaptée pour la résolution de systèmes linéaires creux impliquant des structures de données creuses.

A.2 Méthode du Gradient Conjugué

La difficulté majeure dans l'implantation d'une méthode implicite réside dans la résolution d'un système linéaire creux. En effet un tel système requière des structures de données appropriées permettant la gestion de matrices creuses de grande taille et permettant également d'effectuer les opérations nécessaires sur ces matrices. L'emploi de la Méthode du Gradient Conjugué va permettre d'éviter ces difficultés. Nous reprenons dans cette partie les explications fournies dans un rapport technique par Jonathan Richard Shewchuk en 1994 sur la méthode du Gradient Conjugué [111].

La Méthode du Gradient Conjugué (GC) est une méthode itérative permettant de résoudre des systèmes de grosse taille d'équations linéaires de la forme

$$Ax = b, \tag{A.3}$$

où x est un vecteur inconnu, b un vecteur connu et A une matrice connue, carrée, symétrique définie-positive. Nous rappelons qu'une matrice A est *définie-positive* si, pour tout vecteur non nul x,

$$x^T Ax > 0. \tag{A.4}$$

Les méthodes itératives, telles que le GC, sont fort bien adaptées dans l'emploi de matrices creuses étant performantes notamment en terme de mémoire.

A.2.1 Forme quadratique

Une forme quadratique est une fonction scalaire de la forme

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c, \tag{A.5}$$

où A est une matrice, x et b des vecteurs et c une constante scalaire. Si A est symétrique définie-positive, $f(x)$ est alors minimisée par la solution de $Ax = b$. La méthode du Gradient Conjugué fait en effet partie de la catégorie des méthodes itératives connues comme étant des *méthodes de minimisation*. Jonathan Richard Shewchuk [111] démontre cette idée à l'aide d'un simple exemple, avec

$$A = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}, \quad b = \begin{bmatrix} 2 \\ -8 \end{bmatrix}, \quad c = 0. \tag{A.6}$$

La figure A.1 illustre ce système $Ax = b$. Dans le cas général, la solution x correspond au point d'intersection des n hyperplans, chacun étant de dimension $n-1$. Pour ce problème, la solution x du système $Ax = b$ correspond à l'intersection des deux droites définissant le système, soit $x = [2, -2]^T$.

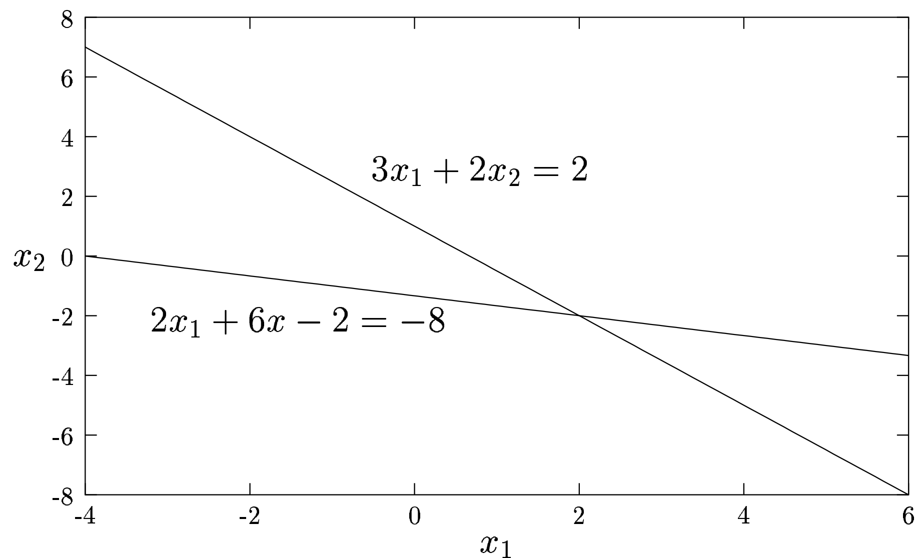


Figure A.1 *Système linéaire à deux dimensions : la solution du système correspond au point d'intersection des deux droites*

La figure A.2 représente le graphe et les courbes de niveau de la forme quadratique $f(x)$ pour la matrice A, le vecteur b et le scalaire c donnés. Le point minimum de la surface définie par $f(x)$ est effectivement le point $(2, -2)$ solution du système linéaire.

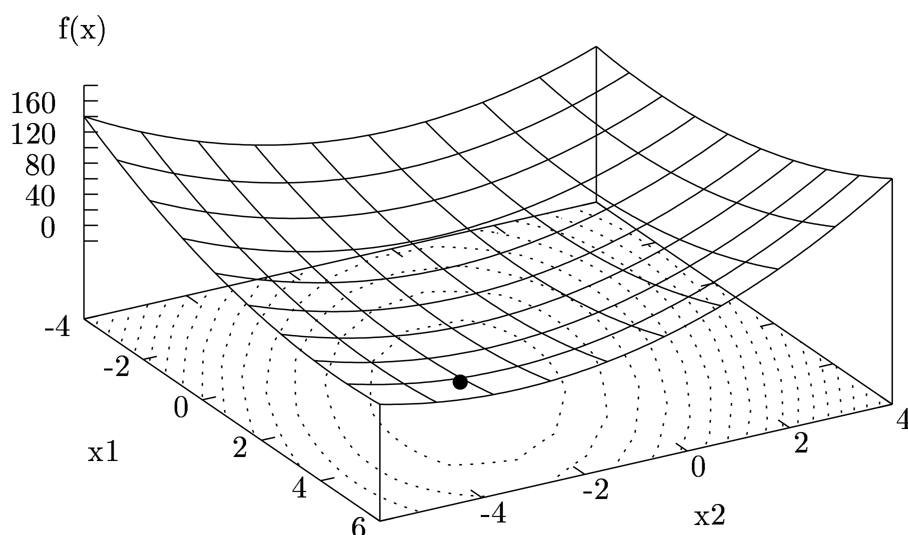


Figure A.2 *Graphe et courbes de niveau de la forme quadratique $f(x)$: le point minimum de cette surface est la solution de $Ax = b$, chaque courbe ellipsoïdale représente un $f(x)$ constant, le point correspond au $f(x)$ minimum*

La forme paraboloïdale de la surface définie par $f(x)$ provient du fait que A est une matrice définie-positive. Le *gradient* de la forme quadratique est définie par

$$f'(x) = \begin{bmatrix} \frac{\partial}{\partial x_0} f(x) \\ \frac{\partial}{\partial x_1} f(x) \\ \vdots \\ \frac{\partial}{\partial x_{n-1}} f(x) \end{bmatrix}. \tag{A.7}$$

Le gradient est un champs de vecteurs qui, pour un point x donné, pointe dans la direction de la valeur la plus forte de $f(x)$. Le gradient est nul au bas de la boule paraboloïdale signifiant que $f(x)$ est minimale quand $f'(x)$ est nulle. La dérivée de la forme quadratique (A.5) est définie par

$$f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b. \tag{A.8}$$

Si A est symétrique, cette équation peut s'écrire sous la forme

$$f'(x) = Ax - b. \tag{A.9}$$

En définissant le gradient nul, nous retombons sur le système linéaire (A.3) que nous cherchons à résoudre. La solution de $Ax = b$ est donc un point critique de $f(x)$, c'est-à-dire un x pour lequel $f'(x)$ est nulle. Par conséquence si A est une matrice symétrique définie-positive, cette solution est un minimum de $f(x)$, et ainsi $Ax = b$ peut être résolu en cherchant un x minimisant $f(x)$.

A.2.2 La méthode de la plus grande pente

Dans la méthode de la plus grande pente, nous commençons à un point arbitraire $x_{(0)}$ et ensuite nous descendons vers le bas de la paraboloïde. Nous obtenons ainsi une série de pas $x_{(1)}, x_{(2)}, \ldots$ qui s'arrête dès que nous considérons être assez proche de la solution x.

Quand nous avançons d'un pas, nous choisissons la direction dans laquelle f décroît le plus rapidement, c'est-à-dire la direction opposée de celle désignée par $f'(x_{(i)})$, soit la direction $-f'(x_{(i)}) = b - Ax_{(i)}$ si nous reprenons l'équation (A.9).

Introduisons quelques définitions utiles pour la compréhension de la méthode. L'*erreur* $e_{(i)} = x_{(i)} - x$ est un vecteur indiquant si nous sommes encore éloignés ou non de la solution. Le *résidu* $r_{(i)} = b - Ax_{(i)}$ indique quant à lui, la distance nous séparant de la valeur correcte b. La relation $r_{(i)} = -Ae_{(i)}$ apparaît alors entre ces deux entités. De plus le résidu peut être exprimé par $r_{(i)} = -f'(x_{(i)})$, c'est-à-dire que le résidu correspond à la direction de la plus grande pente.

Supposons que nous commençons au point $x_{(0)} = [-2, -2]^T$. Le premier pas est effectué le long de la direction de la pente la plus grande, c'est-à-dire en suivant une droite partant du point $x_{(0)}$ et de pente $-f'(x_{(0)})$. En d'autres termes, le point suivant est défini par

$$x_{(1)} = x_{(0)} + \alpha r_{(0)}. \tag{A.10}$$

Reste à choisir la taille du pas que nous effectuons à chaque itération. En fait α est choisi de façon à minimiser f le long de la droite et α minimise f quand la *dérivée directionnelle* $\frac{d}{d\alpha} f(x_{(1)})$ est égale à zéro. Nous obtenons alors

$$\frac{d}{d\alpha} f(x_{(1)}) = f'(x_{(1)})^T \frac{d}{d\alpha} x_{(1)} = f'(x_{(1)})^T r_{(0)}.$$

Nous en déduisons que α est choisi en faisant en sorte que $r_{(0)}$ et $f'(x_{(1)})$ soient orthogonales. Après quelques calculs intermédiaires [111], nous obtenons

$$\alpha = \frac{r_{(0)}^T r_{(0)}}{r_{(0)}^T A r_{(0)}}. \tag{A.11}$$

En résumé, à chaque itération de la méthode de la plus grande pente, nous avons les relations suivantes

$$\begin{aligned} r_{(i)} &= b - Ax_{(i)}, && \text{(A.12)} \\ \alpha_{(i)} &= \frac{r_{(i)}^T r_{(i)}}{r_{(i)}^T A r_{(i)}}, && \text{(A.13)} \\ x_{(i+1)} &= x_{(i)} + \alpha_{(i)} r_{(i)}. && \text{(A.14)} \end{aligned}$$

L'algorithme écrit de cette façon requière deux multiplications matrice-vecteur par itération, il est donc préférable de l'écrire de la façon suivante

$$r_{(i+1)} = r_{(i)} + \alpha_{(i)} A r_{(i)}, \tag{A.15}$$

permettant ainsi de supprimer une des deux multiplications matrice-vecteur fort coûteuses en terme de calcul. La figure A.3 illustre la méthode avec l'exemple décrit précédemment. Nous obtenons un parcours en zigzag provenant du fait que chaque gradient est orthogonal au gradient précédent.

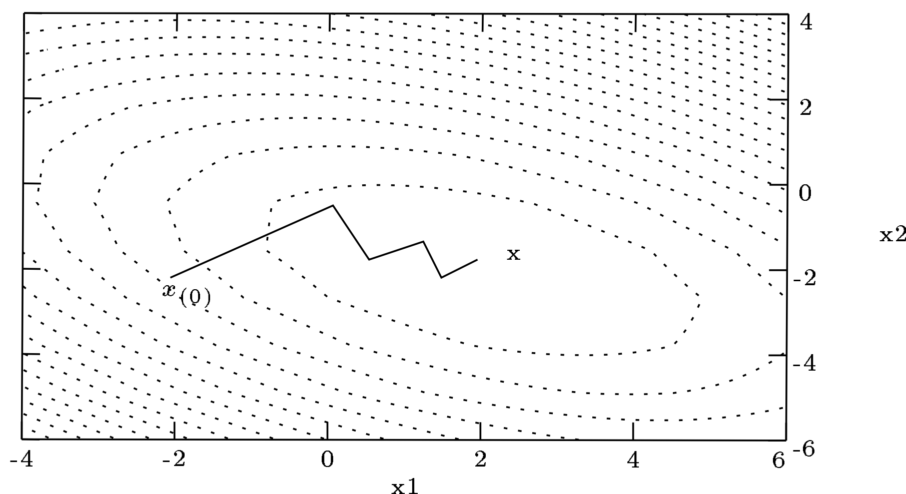


Figure A.3 *Méthode de la plus grande pente commençant à $x_{(0)} = [-2, -2]^T$ et convergeant pour $x = [2, -2]^T$*

A.2.3 La méthode des directions conjuguées

La méthode de la plus grande pente a tendance à ré-effectuer un pas dans une direction déjà prise auparavant (figure A.3), c'est-à-dire qu'elle n'a pas correctement évalué la distance à parcourir dans cette direction la première fois. Introduisons un ensemble de directions orthogonales $d_{(0)}, d_{(1)}, ..., d_{(n-1)}$. Dans chacune de ces directions nous ne voulons effectuer exactement qu'un seul pas.

La figure A.4 illustre cette idée. Le premier pas horizontal permet l'obtention de la coordonnée x_1, et le second pas amène à la solution x. Nous pouvons noter que $e_{(1)}$ est orthogonale à la direction $d_{(0)}$. En définitive, à chaque pas nous choisissons le point

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)}. \tag{A.16}$$

Pour obtenir la valeur de $\alpha_{(i)}$, nous utilisons le fait que $e_{(i+1)}$ doit être orthogonale à $d_{(i)}$ et que nous ne devons ensuite plus refaire de pas dans cette même direction $d_{(i)}$. Ces conditions peuvent se formuler par

$$d_{(i)}^T e_{(i+1)} = d_{(i)}^T \left(e_{(i)} + \alpha_{(i)} d_{(i)}\right) = 0,$$

permettant d'en déduire les valeurs $\alpha_{(i)}$ définies par $\alpha_{(i)} = -\frac{d_{(i)}^T e_{(i)}}{d_{(i)}^T d_{(i)}}$.

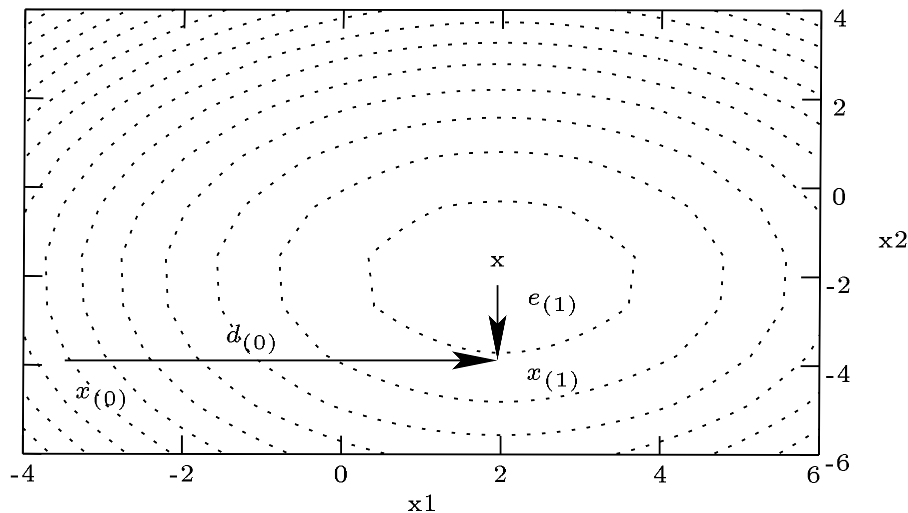


Figure A.4 *La méthode des directions conjuguées*

Pour le moment si nous souhaitons calculer $\alpha_{(i)}$, nous devons connaître $e_{(i)}$. Or le fait de connaître $e_{(i)}$ signifie que le problème a été résolu. La solution réside dans le fait de rechercher des directions *A-orthogonales* et non orthogonales : deux vecteurs $d_{(i)}$ et $d_{(j)}$ sont *A-orthogonaux* ou *conjugués*, si $d_{(i)}^T A d_{(j)} = 0$.

Nous souhaitons désormais que $e_{(i+1)}$ soit A-orthogonal à $d_{(i)}$. Cela revient à chercher le point minimum le long de la direction $d_{(i)}$ comme pour la méthode de la plus grande pente. Nous avons alors comme relation

$$\alpha_{(i)} = -\frac{d_{(i)}^T A e_{(i)}}{d_{(i)}^T A d_{(i)}} = \frac{d_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}}. \tag{A.17}$$

En remplaçant le vecteur de direction par le vecteur de résidu, nous retombons sur la relation observée pour la méthode de la plus grande pente. Il ne reste plus qu'à trouver une méthode permettant de générer l'ensemble des directions A-orthogonales $\{d_{(i)}\}$. Pour cela il existe une méthode nommée *procédé d'orthonormalisation de Gram-Schmidt*.

Soient un ensemble de n vecteurs linéairement indépendants $u_0, u_1, ..., u_{n-1}$. Afin de construire $d_{(i)}$, nous prenons u_i et nous lui ôtons toutes les composantes A-orthogonales par rapport aux précédents vecteurs d. En d'autres termes, nous avons

$$d_{(0)} = u_0, \quad d_{(i)} = u_i + \sum_{k=0}^{i-1} \beta_{ik} d_{(k)}, i > 0. \tag{A.18}$$

où les β_{ik} sont définies pour $i > k$ par

$$\begin{aligned} d_{(i)}^T A d_{(j)} &= u_i^T A d_{(j)} + \sum_{k=0}^{i-1} \beta_{ik} d_{(k)}^T A d_{(j)}, && \text{(A.19)} \\ 0 &= u_i^T A d_{(j)} + \beta_{ij} d_{(j)}^T A d_{(j)}, i > j && \text{(A.20)} \\ \beta_{ij} &= -\frac{u_i^T A d_{(j)}}{d_{(j)}^T A d_{(j)}}. && \text{(A.21)} \end{aligned}$$

Le principal inconvénient de cette méthode réside dans le fait que nous devons garder en mémoire tous les vecteurs anciens afin d'en construire un nouveau, et de plus $\mathcal{O}(n^3)$ opérations sont nécessaires pour générer l'ensemble des vecteurs. Cette méthode devient équivalente à la méthode *d'élimination de Gauss*, si les vecteurs sont construits à partir des vecteurs unitaires.

A.2.4 La méthode des gradients conjugués

La méthode du Gradient Conjugué est comparable à la méthode des directions conjuguées, dans laquelle les vecteurs de directions sont construits à partir des vecteurs de résidu, c'est-à-dire en remplaçant u_i par $r_{(i)}$. Le fait que les vecteurs de résidu soient orthogonaux aux vecteurs de direction précédemment calculés, garantit le fait de construire un nouvelle direction linéairement indépendante à moins que le vecteur de résidu soit nul, et dans ce cas le problème serait déjà résolu. Les vecteurs de résidu étant orthogonaux aux vecteurs de direction précédemment calculés, ils sont également orthogonaux aux précédents vecteurs de résidu

$$r_{(i)}^T r_{(j)} = 0, i \neq j. \tag{A.22}$$

Les constantes de Gram-Schmidt (A.21) sont définies par

$$\beta_{ij} = -u_i^T A d_{(j)} / d_{(j)}^T A d_{(j)}.$$

En utilisant les vecteurs de résidu à la place des vecteurs de directions d, la majorité des termes β_{ij} disparaissent [111], et ainsi il ne devient plus nécessaire de stocker tous les vecteurs précédents afin d'assurer la A-orthogonalité des nouveaux vecteurs construits.

Par conséquent la complexité en temps et en mémoire de la méthode du Gradient Conjugué passe de $\mathcal{O}(n^2)$ à $\mathcal{O}(m)$ avec m le nombre d'éléments non nuls de la matrice A. La méthode des Gradients Conjugués s'exprime au travers des relations suivantes :

$$d_{(0)} = r_{(0)} = b - Ax_{(0)}, \tag{A.23}$$

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}}, \tag{A.24}$$

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)}, \tag{A.25}$$

$$r_{(i+1)} = r_{(i)} + \alpha_{(i)} A d_{(i)}, \tag{A.26}$$

$$\beta_{(i+1)} = \beta_{i+1,i} = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}}, \tag{A.27}$$

$$d_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} d_{(i)}. \tag{A.28}$$

La performance de la méthode des Gradients Conjugués est illustrée par la figure A.5.

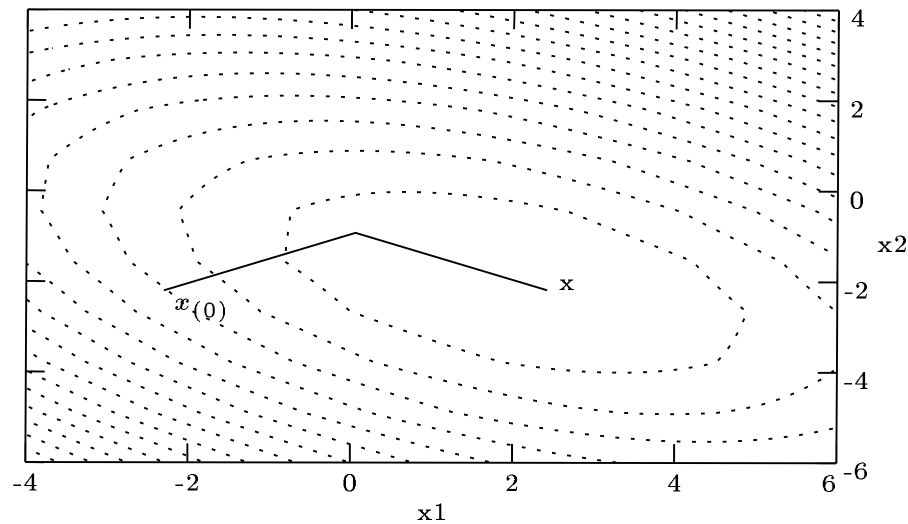


Figure A.5 *La méthode des Gradients Conjugués*

Nous allons résumer toutes ces étapes en étudiant les algorithmes séquentiel et parallèle de la méthode du Gradient Conjugué.

A.3 Algorithmes du Gradient Conjugué

A.3.1 Algorithme séquentiel

L'algorithme de la méthode du Gradient Conjugué itère jusqu'à ce que le facteur d'erreur prenne une valeur en dessous de ε qui traduit la précision souhaitée [109, 75] :

Algorithme 24 Algorithme du Gradient Conjugué pour la résolution de $Ax = b$

```
1 :   β ← 0;            // Initialisation du facteur d'erreur
2 :   x ← 0;            // Initialisation de la solution
3 :   r ← b − Ax;       // Initialisation du vecteur de residu

4 :   Faire
5 :     α ← r^T r;              // Initialisation du pas

6 :     Si (β ≠ 0)
7 :       d ← r + (α/β)d;       // Calcul de la nouvelle direction
8 :     Sinon
9 :       d ← r;                // Initialisation du vecteur de direction

10 :     β ← d^T Ad;           // Calcul du facteur d'erreur
11 :     r ← r − (α/β)Ad;      // Calcul du vecteur de residu
12 :     x ← x + (α/β)d;       // Calcul de la solution
13 :     β ← α;                // Nouveau facteur d erreur

14 :   Jusqu'a (β < ε)         // Iteration jusqu'a la precision souhaitee
```

Dans son rapport technique, Jonathan Richard Shewchuk [111] effectue une analyse très précise sur la convergence de ces trois différentes méthodes de résolution. Nous ne donnons ici que les résultats de cette analyse en terme de complexité algorithmique.

Les opérations dominantes durant une itération de la méthode de la plus grande pente ou de la méthode du Gradient Conjugué sont les produits matrice-vecteur. Considérons le cas d'une matrice creuse de taille $n \times n$ ayant m éléments non nuls. Le produit de cette matrice creuse avec un vecteur nécessite $\mathcal{O}(m)$ opérations avec $m \in \mathcal{O}(n)$.

Supposons que nous souhaitons effectuer un nombre suffisant d'itérations afin de réduire la norme de l'erreur d'un facteur ε, soit $\|e_{(i)}\| \leq \varepsilon \|e_{(0)}\|$. Le nombre maximal d'itérations nécessaires à l'obtention de cette précision en utilisant la méthode de la plus grande pente est

$$i \leq \left\lceil \frac{1}{2} \mathcal{K} \, ln \left(\frac{1}{\varepsilon} \right) \right\rceil ,$$

alors que le nombre maximal d'itérations nécessaires à la méthode du GC est

$$i \leq \left\lceil \frac{1}{2}\sqrt{\mathcal{K}}\, ln\left(\frac{2}{\varepsilon}\right)\right\rceil,$$

avec $\mathcal{K}$ *le nombre spectral* de la matrice A correspondant au rapport entre la plus grande et la plus petite des valeurs propres de la matrice A, soit

$$\mathcal{K} = \frac{\lambda_{max}}{\lambda_{min}}.$$

La complexité en temps de la méthode de la plus grande pente est en $\mathcal{O}(m\mathcal{K})$, alors que la méthode du GC a une complexité en temps en $\mathcal{O}(m\sqrt{\mathcal{K}})$. Les deux algorithmes ont par contre une complexité en mémoire de l'ordre de $\mathcal{O}(m)$.

A.3.2 Algorithme parallèle

L'algorithme du Gradient Conjugué comporte principalement des calculs d'algèbre linéaire de type produits matrice-vecteur ou produits scalaires. De plus dans le cadre de la simulation de textiles, la matrice principale de l'algorithme possède une structure creuse. Notre objectif est donc de réussir à paralléliser le plus efficacement possible la méthode du Gradient Conjugué associée à des structures de données creuses.

Références bibliographiques

De nombreux articles relatent des problèmes liés à la gestion de structure de données creuses, ainsi que ceux rencontrés pour effectuer des calculs d'algèbre linéaire en parallèle. Nous invitons donc le lecteur à se référer aux publications que nous allons décrire brièvement ci-après, pour obtenir de plus amples détails.

Meier et Eigenmann [85] publient une méthode de parallélisation de l'algorithme du Gradient Conjugué dédiée à une machine parallèle multi processeurs comportant une mémoire hiérarchique (Cedar). La matrice A et les vecteurs sont partitionnés en blocs, puis sont distribués sur les processeurs. Chaque processeur effectue ensuite les calculs relatifs à ses données. Des points de synchronisation sont rajoutés dans l'algorithme pour effectuer les produits scalaires. En effet dès qu'un processeur a terminé sa part de calcul pour le produit scalaire il le signale aux autres. Ensuite, quand tous les processeurs ont terminé, ils peuvent lire les valeurs calculées par les autres processeurs et les accumuler pour obtenir le produit scalaire final. Nous pouvons dès lors noter une redondance de calcul dans cette méthode, puisque tous les processeurs effectuent la même somme finale.

Gallivan *et al.* [49, 47, 48] ont publié des articles relatant notamment de techniques permettant d'implanter en parallèle des méthodes de résolution de systèmes linéaires creux non symétriques. Cette parallélisation se base sur une réorganisation des données

de la matrice A permettant l'obtention d'une forme triangulaire supérieure. Une stratégie de placement est alors employée afin de distribuer les données sur les processeurs disponibles. Cette méthode est également dédiée au système parallèle Cedar.

Ujaldon *et al.* [121, 120, 10, 106] fournissent une technique pour le stockage d'une matrice creuse A. Ils utilisent pour cela trois tableaux (format HPF) : le premier tableau stocke les valeurs non nulles de A, le second tableau comporte les numéros de ligne relatifs aux éléments non nuls et le dernier tableau les numéros de colonnes correspondants. La matrice creuse A est au préalable décomposée en plusieurs sous-matrices creuses rectangulaires. Ces sous-matrices peuvent ensuite utiliser la même technique de stockage que pour A. Cette technique de décomposition est employée récursivement pour obtenir autant de sous-matrices que de processeurs. Les indexes des tableaux sont reconstruits au fur et à mesure afin que chaque processeur possède l'ensemble des données nécessaires à l'accès des sous-matrices locales.

Demmel, Heath et van der Vorst, dans leur rapport de recherche relatif à LAPACK intitulé "Parallel numerical linear algebra" [36], présentent les difficultés liées aux calculs d'algèbre linéaire effectués sur différents types de machines parallèles (mémoire partagée, mémoire distribuée). Ils débutent avec un simple produit matrice-vecteur pour ensuite détailler des algorithmes plus compliqués tels que la méthode du Gradient Conjugué, la factorisation de Choleski, ou encore la méthode de Jacobi pour le calcul des valeurs propres d'une matrice. Ils abordent également quelques techniques de restructuration de données pour gérer des matrices creuses.

D'autre part, dans leur livre intitulé "Parallel and distributed computation" [19], Bertsekas et Tsitsiklis présentent brièvement des pistes de parallélisation de l'algorithme du Gradient Conjugué, ainsi que Kumar *et al.* dans leur livre "Introduction to parallel computing" [75].

Idée générale de l'algorithme parallèle

Nous discutons brièvement ici des solutions qui existent pour calculer en parallèle les différents produits scalaires et produits matrice-vecteur que comportent l'algorithme du Gradient Conjugué :

$$\alpha \leftarrow r^T r;\ d \leftarrow r + (\frac{\alpha}{\beta})d;\ \beta \leftarrow d^T Ad;\ r \leftarrow r - (\frac{\alpha}{\beta})Ad;\ x \leftarrow x + (\frac{\alpha}{\beta})d;$$

Supposons qu'au début de la $i^{ème}$ itération, les vecteurs de positions, résidus et de directions, soient $x(t), r(t-1), d(t-1)$, aient déjà été calculés. Nous avons alors à évaluer $Ax(t)$, puis le produit scalaire $r(t)^T r(t)$ pour l'obtention du nouveau pas $\alpha(t)$. Ensuite nous devons calculer le produit scalaire $d(t)^T Ad(t)$ pour obtenir le facteur d'erreur $\beta(t)$ et finalement évaluer $Ad(t)$ pour obtenir le nouveau vecteur de résidu. En négligeant les additions de vecteurs et les multiplications de vecteur par un scalaire, les calculs principaux de l'algorithme sont donc deux multiplications matrice-vecteur et deux produits

scalaires. De plus nous pouvons noter que c'est la même matrice A qui est impliquée dans ces deux multiplications matrice-vecteur.

Prenons le cas des architectures parallèles de passages de messages. Si N processeurs sont disponibles, il est alors naturel de laisser le $i^{ème}$ processeur contrôler la $i^{ème}$ composante des vecteurs à calculer, soient $x(t), r(t)$ et $d(t)$. Les produits scalaires de ces vecteurs sont calculés en laissant le $i^{ème}$ processeur calculer le produit entre les $i^{èmes}$ composantes des vecteurs et ensuite les sommes partielles sont accumulées le long des processeurs. Nous avons alors un seul noeud d'accumulation. Puis les valeurs obtenues pour les différents produits scalaires sont diffusées à tous les processeurs.

Nous supposons désormais que chaque processeur possède les valeurs de lignes différentes de la matrice A. Le produit matrice-vecteur $Ax(t)$ est alors calculé en diffusant le vecteur $x(t)$ au préalable à tous les processeurs et ensuite le $i^{ème}$ processeur peut calculer le produits scalaire entre x et la $i^{ème}$ ligne de A. Il existe également une autre possibilité pour effectuer ce calcul. Le $i^{ème}$ processeur peut calculer $[A]_{ji}x_i$ pour chaque j, et cette quantité peut ensuite être propagée à chaque processeur j, la somme partielle étant effectuée au fur et à mesure.

De plus si moins de N processeurs sont disponibles, la solution est la même sauf qu'il y a alors plus de lignes et plus de composantes de vecteurs qui sont assignées au départ auprès des processeurs.

D'autre part, il ne faut pas oublier que dans notre cas la matrice A est une matrice creuse. La multiplication de A par n'importe quel vecteur est effectuée plus efficacement en utilisant une structure de donnée appropriée exploitant la non densité de la matrice. La limitation de l'efficacité provient alors généralement du réseau à cause des communications à effectuer entre les processeurs. Il y a alors un compromis à trouver entre le nombre minimal de processeurs à utiliser et le nombre de composantes à assigner à chaque processeur afin que le nombre de calculs à effectuer sur chaque processeur soit comparable aux communications.

De plus amples détails en ce qui concerne nos choix de parallélisation de la méthode du Gradient Conjugué sont fournis au sein du chapitre 9.

A.4 Conclusion

La méthode du Gradient Conjugué est une méthode efficace permettant la résolution de systèmes comportant des matrices symétriques définies-positives. C'est une méthode itérative non-stationnaire. Les méthodes non-stationnaires diffèrent des méthodes stationnaires dans le fait que les calculs nécessitent des informations qui sont modifiées à chaque itération.

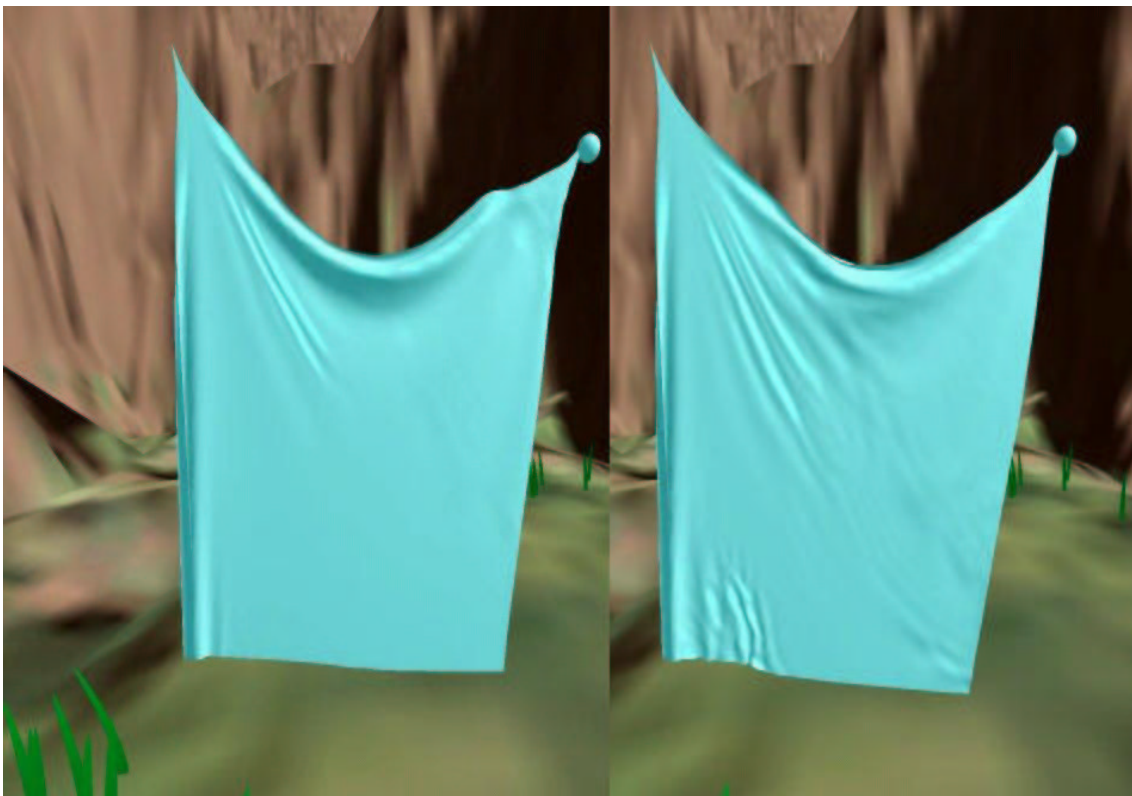
La méthode du Gradient Conjugué procède en générant des séquences de vecteurs correspondants aux itérations (c'est-à-dire aux approximations successives de la solution),

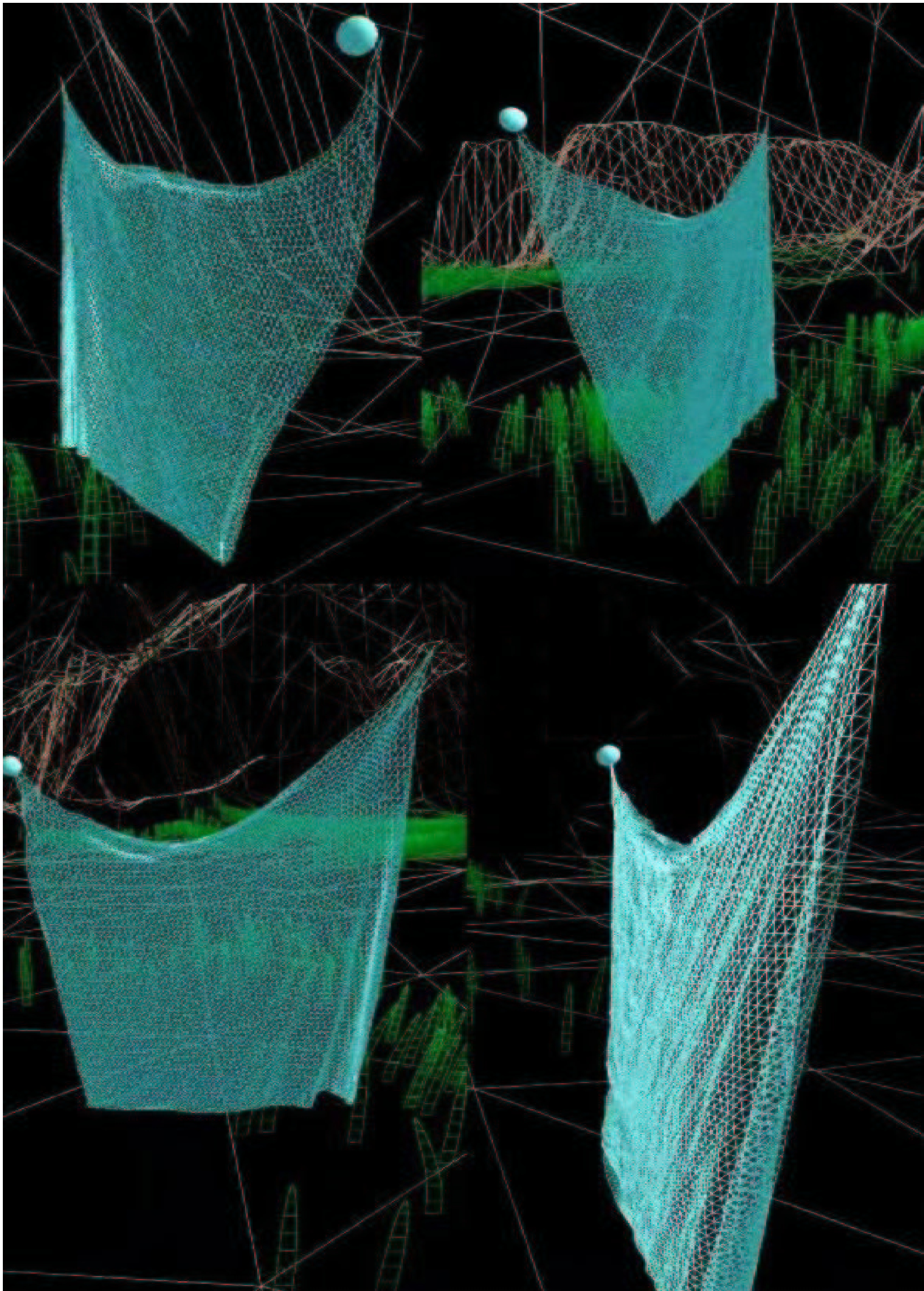
des vecteurs de résidu correspondant aux itérations et des vecteurs de directions utilisés pour mettre à jour la solution et le résidu. Malgré le fait que la taille de ces séquences peut devenir importante, seul un petit nombre de vecteurs doit être gardé en mémoire. A chaque itération de la méthode, deux produits scalaires sont effectués pour calculer les nouveaux scalaires définissant la séquence qui doit satisfaire quelques contraintes d'orthogonalité. Dans le cas de systèmes linéaires définis-positifs, ces conditions impliquent le fait que la distance à la solution exacte est minimisée par une norme.

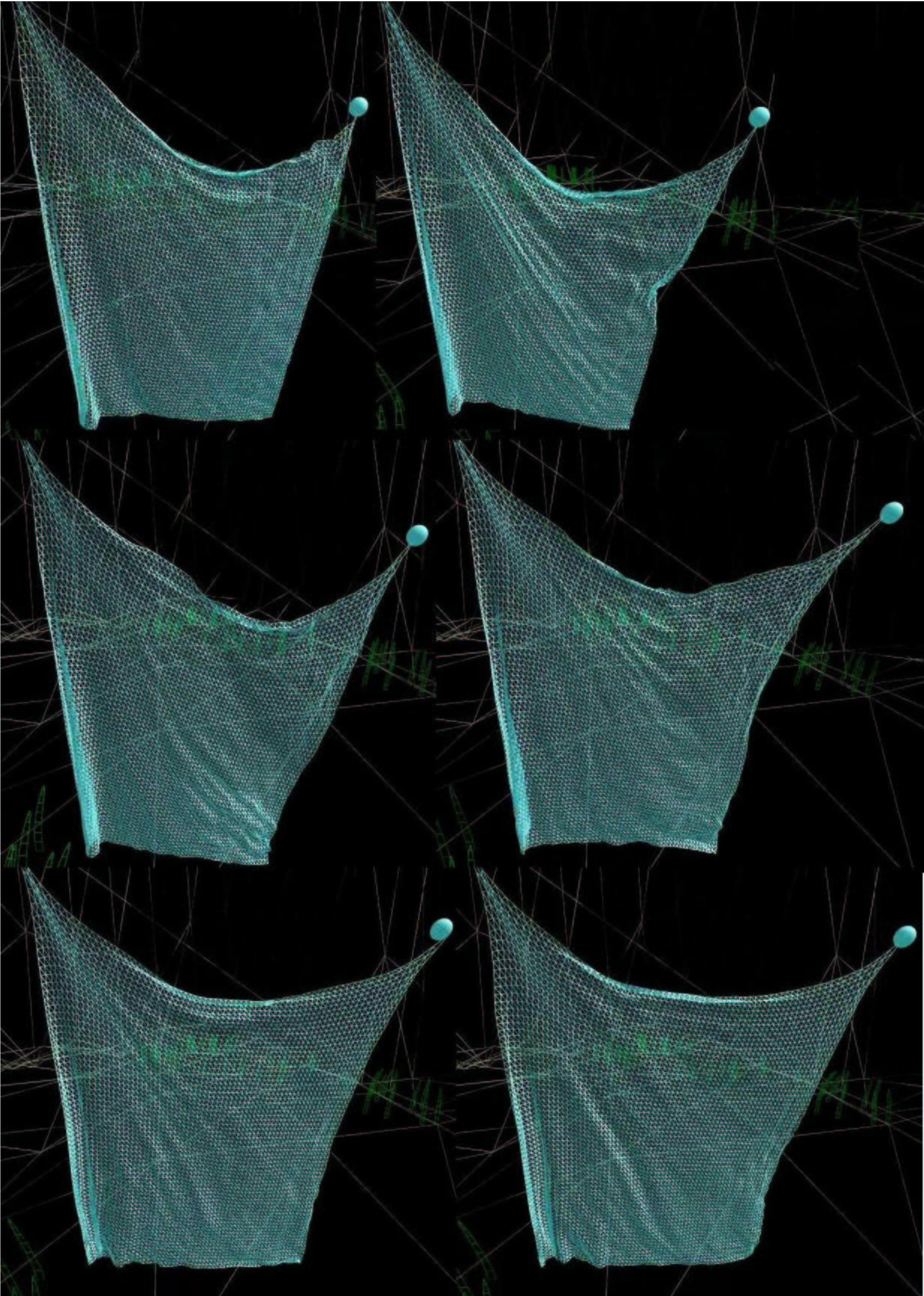
Au niveau parallélisme, l'algorithme du Gradient Conjugué comporte plusieurs points de synchronisation qui ne facilitent pas l'obtention d'une accélération optimale. Il y a des compromis à faire lors du placement des données sur les processeurs afin de ne pas engendrer un nombre trop important de communications par rapport aux calculs.

Galerie d'images B

Enfin pour finir quelques images issues de la simulation d'un tissu comportant 490 particules dont l'une des extrémités est fixée tandis que l'autre est contrôlée par l'utilisateur...







Bibliographie

[1] A. Adamson, K.Bielenberg, G. Bruder, J. Buhler, J. Hegedus, A. Lamorlette, L. Modesto, S. Singer, S. Peterson, V. Rangaraju, R. Vogt, and M. Wendell. "shrek" : The story behind the screen. In *SIGGRAPH 2001*, Los Angeles, CA, USA, August 2001.

[2] G. A. Agha. *ACTORS : a model of concurrent computations in distributed systems*. Computer and communication science, University of Michigan, USA, 1985.

[3] G. A. Agha. *ACTORS : a model of concurrent computations in distributed systems*. MIT Press, 1990.

[4] S. Ahuja, N. Carriero, D. Gelertner, and V. Krishnaswamy. Matching language and hardware for parallel computation in the linda machine. *IEEE Transactions on Computers*, 1988(8) :921–929, August 37.

[5] J. Allard, V. Gouranton, L. Lecointre, E. Melin, and B. Raffin. Net juggler : Running VR juggler with multiple displays on a commodity component cluster. In *IEEE VR*, pages 275–276, Orlando, USA, March 2002.

[6] J. Allard, V. Gouranton, E. Melin, and B. Raffin. Parallelizing Pre-rendering Computations on a Net Juggler PC Cluster. In *Immersive Projection Technology Symposium*, Orlando, USA, March 2002.

[7] J. Allard, L. Lecointre, V. Gouranton, E. Melin, and B. Raffin. Net Juggler Guide. Technical Report RR-LIFO-2001-02, LIFO, Orléans, France, June 2001, http ://netjuggler.sourceforge.net.

[8] J. Allard, B. Raffin, and F. Zara. Coupling Parallel Simulation and Multi-Display Visualization on a PC Cluster. In *International Conference on Parallel and Distributed, Euro-Par 2003*, Klagenfurt, Austria, August 2003.

[9] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1) :3–15, 1997.

[10] R. Asenjo, L.F. Romero, M. Ujald'on, and E.L. Zapata. Sparse Block and Cyclic Data Distributions for Matrix Computations. *High Performance Computing : Technology, Methods and Applications*, pages 359–377, 1995.

[11] P. Augerat, C. Goudeseune, H. Kaczmarski, B. Raffin, B. Schaeffer, L. Soares, and M. K. Zuffo. Commodity clusters for immersive projection environments. In *Siggraph 2002 Course Notes*, San Antonio, Texas, USA, July 2002.

[12] G. Authié, J-M. Garcia, A. Ferreira, J-L. Roch, G. Villard, J. Roman, C. Roucairol, and B. Virot. *Parallélisme et applications irrégulières*. Hermès, 1995.

[13] D. Baraff and A. Witkin. Large Steps in Cloth Simulation. In Michael Cohen, editor, *SIGGRAPH'98 Conference Proceedings*, Annual Conference Series, pages 43–54, USA, 1998. ACM SIGGRAPH, Addison Wesley.

[14] D. Baraff, A. Witkin, and M. Kass. Untangling Cloth. *SIGGRAPH'03 Conference Proceedings*, 22(3) :862–870, 2003.

[15] M. Benzi, J. K. Cullum, and M. Tuma. Robust approximative inverse preconditioning for the Conjugate Gradient Method. *SIAM J. Sci. Comput.*, 22(4) :1318–1332, 2000.

[16] P.-E. Bernard. *Parallélisation et multiprogrammation pour une application irrégulière de dynamique moléculaire opérationnelle*. Mathématiques appliquées, Institut National Polytechnique de Grenoble, France, Octobre 1997.

[17] P-E. Bernard, T. Gautier, and D. Trystram. Large scale simulation of parallel molecular dynamics. In *Proceedings of Second Merged Symposium IPPS/SPDP 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico, April 1999.

[18] P.-E. Bernard, B. Plateau, and D. Trystram. Using threads for developing parallel applications : Molecular dynamics as a case study. In Trobec, editor, *Parallel Numerics*, pages 3–16, Gozd Martuljek, Slovenia, September 1996.

[19] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation : Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[20] K. S. Bhat, C. D. Twigg, J. K. Hodgins, P. K. Khosla, Z. Popovic, and S. M. Seitz. Estimating Cloth Simulation Parameters from Video. In *Eurographics / ACM SIGGRAPH Symposium on Computer Animation*, San Diego, California, USA, July 2003. http ://graphics.cs.cmu.edu/projects/clothparameters.

[21] A. Bierbaum, A. Bierbaum, and C. Cruz-Neira. ClusterJuggler : A modular architecture for immersive clustering. In *Commodity Cluster for Virtual Reality 2003, VR 2003 Workshop*, Los Angeles, CA, USA, March 2003.

[22] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. Vr juggler : A virtual platform for virtual reality application development. In *IEEE VR 2001*, Yokohama, Japan, March 2001.

[23] R. Bigliani and J. W. Eischen. Collision Detection in Cloth modelling. *Cloth and Clothing in Computer Graphics*, 1999.

[24] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk : An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1) :55–69, 1996.

[25] D. Breen, D. House, and P. Getto. A particle-based model for simulating the draping behavior of woven cloth. *Textile Research Journal, Vol. 64*, 11 :663–685, November 1994.

[26] J. Briat, I. Ginzburg, and M. Pasin. *Athapascan-0 User Manual*. Projet APACHE, Grenoble, http ://www-id.imag.fr/Logiciels/ath0/.

[27] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. In *SIGGRAPH 2002*, San Antonio, Texas, USA, July 2002.

[28] K-J. Choi and H-S. Ko. Stable but responsible cloth. In *SIGGRAPH 2002*, San Antonio, Texas, USA, July 2002.

[29] J. D. Cohen, M. C. Lin, D. Manocha, and M. Ponamgi. I-COLLIDE : An Interactive and Exact Collision Detection System for Large-Scale Environments. In *Symposium on Interactive 3D Graphics*, pages 189–196, 1995.

[30] F. Cordier and N. Magnenat-Thalmann. Real-time animation of dressed virtual humans. In G. Drettakis and H.-P. Seidel, editors, *EUROGRAPHICS 2002*, Saarbrucken, Germany, September 2002. Blackwell Publishers.

[31] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The Cave Audio VIsual Experience Automatic Virtual Environement. *Communication of the ACM*, 35(6) :64–72, 1992.

[32] G. Dahlquist. Stability and Error Bounds in the Numerical Integration of Ordinary Differential Equations. *Trans. Roy. Inst. Technol.*, (130), 1959.

[33] J. Chassin de Kergommeaux and B. de Oliveira Stein. Flexible performance visualization of parallel and distributed applications. *Future Generation Computer Systems*, 19(5) :735–748, 2003.

[34] J. Chassin de Kergommeaux, B. de Oliveira Stein, and P.-E. Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10) :1253–1274, August 2000.

[35] J. Chassin de Kergommeaux, C. Guilloud, and B. de Oliveira Stein. Flexible Performance Debugging of Parallel and Distributed Applications. In *International Conference on Parallel and Distributed, Euro-Par 2003*, Klagenfurt, Austria, August 2003.

[36] J. Demmel, M. Heath, and H. van der Vorst. Parallel numerical linear algebra. In *Acta Numerica 1993*, pages 111–198. Cambridge University Press, Cambridge, UK, 1993.

[37] M. Desbrun, M. Meyer, and A. H. Barr. Interactive animation of cloth-like objects for virtual reality. *Journal of Vizualisation and Computer Animation*, 2000, http ://www-imagis.imag.fr/Publications/2000/MDDB00.

[38] M. Desbrun, P. Schröder, and A. Barr. Interactive Animation of Structured Deformable Objects. In *Graphics Interface 99*, pages 1–8, 1999.

[39] B. Eberhardt, O. Etzmuss, and M. Hauth. Implicit-Explicit Schemes for Fast Animation with Particle Systems. In D. Thalmann N. Thalmann-Magnenat and B. Arnaldi, editors, *Eurographics Workshop on Computer Animation and Simulation (EGCAS-00)*, pages 137–154. Springer, 2000.

[40] B. Eberhardt, A. Weber, and W. Strasser. A fast, flexible, particle-system model for cloth. *IEEE Computer Graphics and Applications*, 16 :52–59, 1996.

[41] B. Fagin and A. Despain. The performance of parallel prolog programs. *IEEE Transactions on Computers*, 39(12) :1434–1445, 1990.

[42] L. G. L. Fernandes. *Parallélisation d'un algorithme d'appariement d'images quasi-dense*. Informatique : Systèmes et communications, Institut National Polytechnique de Grenoble, France, July 2002.

[43] M.-J. Flynn. Somme Computer Organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9) :948–960, 1972.

[44] I. Foster. *Designing and building parallel programs*. Addison Wesley, 1994.

[45] F. Galilée. *Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, September 1999.

[46] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1 : On-line building data flow graph in a parallel language. In IEEE, editor, *Pact'98*, Paris, France, oct 1998.

[47] K. A. Gallivan, B. A. Marsolf, and H. A. G. Wijshoff. Solving large nonsymmetric sparse linear systems on Cedar. Technical Report 1313, Center for Supercomputing Research and Development, Urbana, IL, USA, 1993.

[48] K. A. Gallivan, B. A. Marsolf, and H. A. G. Wijshoff. The parallel solution of nonsymmetric sparse linear systems using the H^* reordering and an associated factorization. In *Proceeding of the 1994 International Conference on Supercomputing*, pages 419–430, Manchester, England, July 1994.

[49] K. A. Gallivan and Z. Zlatev. Solving general sparse linear systems using conjugate gradient-type methods. In *Proceeding of the 1990 International Conference on Supercomputing*, pages 132–139, Amsterdam, The Netherlands, June 1990. ACM Press.

[50] T. Gautier and J.-L. Roch. Irregular algorithms and on-line scheduling. In INRIA, editor, *Proc. of Stratagem'96, Symposium on parallel computing for solving large scale and irregular applications*, pages 17–37, Sophia-Antipolis, France, jul 1996.

[51] A. Geist, A.Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[52] G. A. Geist, J. A. Kohla, and P. M. Papadopoulos. PVM and MPI : A Comparison of Features. *Calculateurs Paralleles*, 8(2) :137–150, 1996.

[53] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree : A hierarchical Structure for Rapide Interference Detection. *Computer Graphics*, 30 :171–180, 1996.

[54] M. Grobe. *The architecture and use of the Origin 2000*. Academic Computing Services, University of Kansas, January 1998.

[55] Cilk-Cilk Is Supercomputing Technologies Group. *Cilk 5.3 – Reference Manual*. MIT Laboratory for Computer Science, November 2001, http ://supertech.lcs.mit.edu/cilk.

[56] L. J. Guibas, D. Hsu, and L. Zhang. H-walk : hierarchical distance computation for moving convex bodies. In *Proceedings of the fifteenth annual symposium on computational geometry*, pages 265–273, Miami, Floride, USA, January 1999. Association for Computing Machinery.

[57] E. Hairer, C. Lubich, and G. Wanner. Geometric numerical integration illustrated by the Störmer/Verlet method. In *Acta Numerica 2003*, pages 1–51. Cambridge University Press, 2003.

[58] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems.*, volume 8 of *Springer Series in Comput. Mathematics*. Springer-Verlag, New York, NY, USA, second revised edition, 1993.

[59] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems.*, volume 14 of *Springer Series in Comput. Mathematics*. Springer-Verlag, New York, NY, USA, 1996.

[60] E. Hairer and G. Wanner. Stiff differential equations solved by Radau methods. *Journal of Computational and Applied Mathematics*, 111(1) :93–111, 1999.

[61] R. H. Halstead. Parallel symbolic computing. *Computer*, 19(8) :35–43, 1986.

[62] M. Hauth and O. Etzmuss. A high performance solver for the animation of deformable object using advanced numerical methods. In *European Association for Computer Graphics (EUROGRAPHICS'2001)*, Manchester, UK, September 2001. ACM.

[63] M. Hauth, O. Etzmuss, B. Eberhardt, R. Klein, R. Sarlette, M. Sattler, K. Daubert, and J. Kautz. Cloth animation and rendering. In *EUROGRAPHICS 2002*, Saarbrucken, Germany, September 2002. The Eurographics Association. Tutorial.

[64] D. House and D. Breen. *Cloth Modeling and Animation*. A K Peters, Natick, Massachusetts, 2000.

[65] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrathan. WireGL A Scalable Graphics System for Clusters. In *SIGGRAPH'01 Conference Proceedings*, Annual Conference Series, Los Angeles, CA, USA, August 2001. ACM SIGGRAPH, http ://graphics.stanford.edu/software/wiregl.

[66] J. JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[67] C. F. Joerg. The cilk system for parallel multithreaded computing. Technical Report MIT/LCS/TR-701, MIT, 1996.

[68] C. Just, A. Bierbaum, A. Baker, and C. Cruz-Neira. Vr juggler : A framework for virtual reality development. In *2nd Immersive Projection Technology Workshop (IPT98)*, Ames, USA, May 1998.

[69] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1) :96–129, 1998.

[70] G. Karypis and V. Kumar. Multilevel k -way hypergraph partitioning. In *Design Automation Conference*, pages 343–348, 1999.

[71] Z. Kačić-Alesić, M. Nordenstam, and D. Bullock. A practical dynamics system. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 7–16, San Diego, CA, USA, July 2003. Eurographics Association.

[72] S. Kawabata. The standardization and analysis of hand evaluation. Technical report, The textile machinery society of Japan, Osaka, 1980.

[73] M. J. Kilgard. *The OpenGL Utility Toolkit (GLUT) Programming Interface*, November 1996.

[74] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1) :21–36, 1998.

[75] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing, design and analysis of algorithms*. Benjamin/Cummings, 1994.

[76] R. Lario, C. García, M. Prieto, and F. Tirado. Rapid parallelization of a multilevel cloth simulator using openmp. In *Third European Workshop on OpenMP (EWOMP'01)*, Barcelona, Spain, September 2001.

[77] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha. Fast proximity queries with swept sphere volumes. Technical Report TR99-018, Department of Computer Science, Chapel Hill, 1999.

[78] O. S. Lawlor and L. V. Kalé. A voxel-based parallel collision detection algorithm. In *Proceedings of the International Conference in Supercomputing*, pages 285–293. ACM Press, June 2002.

[79] M. C. Lin and S. Gottschalk. Collision Detection between Geometric Models : A Survey. In *Proceedings of IMA Conference on Mathematics of Surfaces*, Birmingham, August and September 1998.

[80] J. Louchet, M. Boccara, D. Crochemore, and X. Provot. Building new tools for synthetic image animation by using evolutionary techniques. In *Evolution Artificielle*, Brest, France, 1995.

[81] J. Louchet, X. Provot, and D. Crochemore. Evolutionary identification of cloth animation models. In Dimitri Terzopoulos and Daniel Thalmann, editors, *Computer Animation and Simulation'95*, pages 44–54. Springer-Verlag, 1995.

[82] N. Magnenat-Thalmann. Efficient Self-Collision Detection on Smoothly Discretized Surface Animations using Geometrical Shape Regularity. *Computer Graphics Forum*, 13(3) :155–166, 1994.

[83] C. Martin. *Déploiement et contrôle d'applications parallèles sur grappe de grande taille*. Informatique : Systèmes et logiciels, Institut National Polytechnique de Grenoble, France, October 2003.

[84] M. Mascaro, A. Mir, and F. Perales. Elastic deformations using finite element methods in computer graphic publications. In H. H. Nagel and F. J. Perales, editors, *LNCS*, volume 1899, pages 38–47, 2000.

[85] U. Meier and R. Eigenmann. Parallelization and performance of conjugate gradient algorithms on the cedar hierarchical-memory multiprocessor. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26, pages 178–188, Williamsburg, VA, April 1991.

[86] B. Mirtich. VClip : Fast and Robust Polyhedral Collision Detection. *ACM Transactions on Graphics*, 17(3) :177–208, 1998.

[87] T. Moller. A fast triangle-triangle intersection test. *JGTOOLS : Journal of Graphics Tools*, 2, 1997.

[88] F. Neyret. *OpenGL : principe et astuces*, 1998, http ://w3imagis.imag.fr/Membres/Fabrice.Neyret/doc/opengl.html. Document issu du groupe de travail iMAGIS du 10/07/98.

[89] O. Nocent, J.M. Nourrit, and Y. Remion. Vers du niveau de détail mécanique pour l'animation dynamique d'objets continus : Application aux textiles tricotés. In *AFIG'00 Association Française d'Informatique Graphique*, Grenoble, France, 2000.

[90] E. C. Orson. Cluster Juggler - PC cluster virtual reality. Master's thesis, Iowa State University, USA, 2002.

[91] P. Palmer, A. Mir, and M. Gonzalez. Stability and complexity study of animated elastically deformable objects. In H. H. Nagel and F. J. Perales, editors, *LNCS*, volume 1899, pages 58–71, 2000.

[92] D. Parks and D. Forsyth. Improved integration for cloth simulation. In *EUROGRAPHICS 2002*, Saarbrucken, Germany, September 2002. I. Navazo Alvaro and Ph. Slusallek. Short Presentations.

[93] F. Pellegrini. Scotch and libscotch 3.4 user's guide. Technical Report 1264-01, LABRI, URM CNRS 5800, November 2001.

[94] F. Pellegrini and J. Roman. Scotch : A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proceedings of HPCN'96*, pages 493–498, Brussels, Belgium, April 1996. Springer.

[95] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C : The Art of Scientific Computing*, chapter 16. Integration of Ordinary Differential Equations. Cambridge University Press, Cambridge, UK, second edition, 1993, http ://www.nr.com/.

[96] X. Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *Graphics Interface '95*, pages 147–154, 1995.

[97] X. Provot. *Animation réaliste de vêtements*. Mathématiques et informatique, Université René Descartes - Paris V, France, December 1997.

[98] X. Provot. Collision and self-collision handling in cloth model dedicated to design garments. In *Graphics Interface '97*, pages 177–189, 1997.

[99] B. Raffin. Des grappes de PC pour la realite virtuelle. In *Imagina 2002*, Monaco, France, February 2002. Invited Talk.

[100] W. T. Reeves. Particle Systems – A Technique for Modeling a Class of Fuzzy Objects. *Proceedings of ACM SIGGRAPH'83, also published as ACM Computer Graphics*, 17(3) :359–376, July 1983.

[101] W. T. Reeves and R. Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *Proceedings of ACM SIGGRAPH'85*, 19(3), July 1985.

[102] R. Revire, F. Zara, and T. Gautier. Efficient and Easy Parallel Implementation of Large Numerical Simulations. In *ParSim 2003 (Special Session of EuroPVM/MPI 2003)*, Venice, Italy, September 2003.

[103] J-L. Roch. Complexité parallèle et algorithmique PRAM. In *Ecole d'automne Conception et Analyse d'Algorithmes Paralleles (CAPA 93)*, Port d'Albret, Landes, France, Septembre 1993.

[104] J.-L. Roch, T. Gautier, and R. Revire. Athapascan : Api for asynchronous parallel programming user's guide. Technical Report RR-0276, INRIA Rhône-Alpes, projet APACHE, February 2003.

[105] L. Romero, E. Zapata, and J. Ramos. Efficient Parallel Solution of a Semiconductor Laser Array Dynamics Model. In *High-Performance Computing and Networking (HPCN Europe'96)*, Brussel, Belgium, April 1996.

[106] L. F. Romero and E. L. Zapata. Data Distributions for Sparse Matrix Vector Multiplication. *Parallel Computing*, 21(4) :583–605, 1995.

[107] S. Romero, L. F. Romero, and E. L. Zapata. Fast Cloth Simulation with Parallel Computers. In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 491–499, Munich, Germany, August/September 2000. Springer-Verlag Heidelberg.

[108] S. Romero, L. F. Romero, E. L. Zapata, and L. Emilio. Parallel Algorithm for Fast Cloth Simulation. In *VECPAR 2000 4th VECtor and PARallel processing*, pages 529–535, Portugal, June 2000.

[109] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[110] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. *CRPC Parallel Computing Handbook*, 2000.

[111] J. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical Report CMU-CS-TR-94-125, Carnegie Mellon University, 1994.

[112] D. Shreiner, editor. *OpenGL Reference Manual : The Official Reference Document to OpenGL*. Addison-Wesley Publishing Company, 3rd edition, December 1999.

[113] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2) :123–169, 1998.

[114] 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*, pages I :11–14, Oconomowoc, WI, 1995.

[115] A. Tanembaum. *Modern Operating Systems*. Prentice Hall, 1992.

[116] Athapascan-1 team. *Athapascan-1*. Projet APACHE, Grenoble, http ://www-id.imag.fr/Logiciels/ath1/.

[117] D. Terzopolous, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. *Proceedings of SIGGRAPH 87*, 21(4) :205–214, July 1987.

[118] D. Terzopoulos, A. Witkin, and M. Kass. Constraints on deformable models : Recovering 3d shape and nonrigid motion. *Artificial Intelligence*, 36 :91–123, November 1988.

[119] R. Thakur, E. Lusk, and W. Gropp. I/O in parallel applications : The weakest link. *The International Journal of High Performance Computing Applications*, 12(4) :389–395, Winter 1998.

[120] M. Ujaldon, S. Sharma, J. Saltz, and E. Zapata. Run-time techniques for parallelizing sparse matrix problems. In *Workshop on Irregular Problems*, 1995.

[121] M. Ujaldon, E.L. Zapata, B.M. Chapman, and H.P. Zima. New Data-Parallel Language Features for Sparse Matrix Computations. In *9th IEEE Int'l. Parallel Processing Symposium*, pages 742–749, Santa Barbara, CA, USA, April 1995.

[122] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast Cloth Animation on Walking Avatars. In *European Association for Computer Graphics (EUROGRAPHICS'2001)*, Manchester, UK, September 2001. ACM.

[123] M. Vilayannur, M. Kandemir, and A. Sivasubramaniam. Kernel-level caching for optimizing i/o by exploiting inter-application data sharing. In *IEEE International Conference on Cluster Computing (CLUSTER'02)*, Chicago, Illinois, USA, September 2002.

[124] P. Volino, M. Courchesne, and N. Magnenat Thalmann. Versatile and Efficient Techniques for Simulating Cloth and Other Deformable Objects. *Computer Graphics*, 29(Annual Conference Series) :137–144, 1995.

[125] P. Volino and N. Magnenat-Thalmann. Interactive Cloth Simulation : Problems and Solutions. In *JWS'97*, 1997.

[126] P. Volino and N. Magnenat-Thalmann. Accurate Collision Response on Polygonal Meshes. In *CA'00 Computer Animation 2000*, Geneva, June 2000.

[127] P. Volino and N. Magnenat-Thalmann. Implementing Fast Cloth Simulation with Collision Response. In *CGI'00 Computer Graphics International*, Geneva, June 2000.

[128] P. Volino and N. Magnenat-Thalmann. Comparing Efficiency of Integration Methods for Cloth Simulation. In *Computer Graphics International 2001 (CGI'01)*, Hong-Kong, China, July 2001.

[129] D. W. Walker and J. J. Dongarra. MPI : A message-passing interface standard. *Supercomputer*, 12(1) :56–68, 1996.

[130] A. Witkin and D. Baraff. Differential Equations Basics. In *SIGGRAPH93 20th International Conference on Computer Graphics and Interactive Techniques*, pages B1–B8, California, USA, August 1993.

[131] A. Witkin and D. Baraff. Physically Based Modeling : Principles and Practice. In *SIGGRAPH'97 Course Notes*, California, USA, August 1997.

[132] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide : The Official Guide to Learning OpenGL.* Addison-Wesley Pub Co, 3rd edition, August 1999.

[133] F. Zara. Simulation physique de textiles sur grappe de processeurs. In *AFIG 2001*, Limoges, France, November 2001.

[134] F. Zara, F. Faure, and J-M. Vincent. Physical cloth simulation on a PC cluster. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 105–112, Blaubeuren, Germany, September 2002. Eurographics Association.

[135] F. Zara, F. Faure, and J-M. Vincent. Parallel Simulation of Large Dynamic System on a PCs Cluster : Application to Cloth Simulation. *Special issue on Cluster/Grid Computing of the International Journal of Computers and Applications (IJCA)*, March 2004.

[136] F. Zara, J-M. Vincent, and F. Faure. Coupling Parallel Simulation and Parallel Visualization on PC Clusters. In *Commodity Cluster for Virtual Reality 2003, VR 2003 Workshop*, Los Angeles, USA, March 2003.

[137] D. Zhang and M. M. F. Yuen. Collision Detection for Clothed Human Animation. In *Pacific Graphics 2000 (PG'00)*, Hong Kong, China, October 2000.

[138] M. Zirkel, V. Markl, and R. Bayer. Exploitation of Pre-sortedness for Sorting in Query Processing : The TempTris-Algorithm for UB-Trees. In M. Adiba, C. Collet, and B. C. Desai, editors, *Proceedings of 2001 International Database Engineering & Applications Symposium (IDEAS'01)*, pages 155–166, Grenoble, France, July 2001. IEEE Computer Society.

Publications

Journaux internationaux

– F. Zara, F. Faure, and J-M. Vincent. Parallel Simulation of Large Dynamic System on a PCs Cluster : Application to Cloth Simulation. *Special issue on Cluster/Grid Computing of the International Journal of Computers and Applications (IJCA)*, March 2004.

Conférences internationales

– F. Zara, F. Faure, and J-M. Vincent. Physical cloth simulation on a PC cluster. In X. Pueyo D. Bartz and E. Reinhard, editors, *Fourth Eurographics Workshop on Parallel Graphics and Visualization 2002*, pages 105–112, Blaubeuren, Germany, September 2002.
– F. Zara, J-M. Vincent, and F. Faure. Coupling Parallel Simulation and Parallel Visualization on PC Clusters. In *Commodity Cluster for Virtual Reality 2003, VR 2003 Workshop*, Los Angeles, USA, March 2003.
– J. Allard, B. Raffin, and F. Zara. Coupling Parallel Simulation and Multi-Display Visualization on a PC Cluster. In *International Conference on Parallel and Distributed, Euro-Par 2003*, Klagenfurt, Austria, August 2003. (Une démonstration du couplage a été réalisée durant la conférence).
– R. Revire, F. Zara, and T. Gautier. Efficient and Easy Parallel Implementation of Large Numerical Simulations. In *ParSim 2003 (Special Session of EuroPVM/MPI 2003)*, Venice, Italy, September 2003.

Conférences nationales

– F. Zara. Simulation physique de textiles sur grappe de processeurs. Dans *Association Française d'Informatique Graphique AFIG 2001*, Limoges, Novembre 2001.

Posters

– F. Zara. Algorithmes parallèles de simulation physique pour la synthèse d'images : Application à l'animation de textiles. 13es *Rencontres Régionales de la Recherche de la Région Rhône-Alpes,* 1er *prix de la catégorie Sciences de l'Ingénieur*, Grenoble, Septembre 2002.

Démonstrations

– F. Zara. Algorithmes parallèles de simulation physique pour la synthèse d'images : Application à l'animation de textiles. *Manifestation pour les 50 ans de l'informatique à Grenoble*, Grenoble, du 23 novembre au 2 décembre 2002.
– F. Zara. Algorithmes parallèles de simulation physique pour la synthèse d'images : Application à l'animation de textiles. *Manifestation pour les 10 ans de l'INRIA Rhône-Alpes*, Montbonnot Saint-Martin, 19 décembre 2002.
– J. Allard, B. Raffin, and F. Zara. Coupling Parallel Simulation and Multi-Display Visualization on a PC Cluster. In *International Conference on Parallel and Distributed, Euro-Par 2003*, Klagenfurt, Austria, August 2003.
– F. Zara. Algorithmes parallèles de simulation physique pour la synthèse d'images : Application à l'animation de textiles. *Fêtes de la Science 2003*, Grenoble, octobre 2003.
– J. Allard, B. Raffin, and F. Zara. Coupling Parallel Simulation and Multi-Display Visualization on a PC Cluster. In *SuperComputing 2003*, Phoenix, USA, November 2003.

Résumé

Algorithmes parallèles de simulation physique pour la synthèse d'images :
application à l'animation de textiles

Cette thèse combine le calcul haute performance à la réalité virtuelle par son apport de méthodes de calcul parallèle pour l'animation d'objets 3D en synthèse d'image. Son application vise plus particulièrement le domaine de la simulation de textiles par modèles physiques. Les lois fondamentales de la dynamique ont en effet été employées pour modéliser le mouvement de plusieurs objets dans un souci de réalisme. Les modèles employés étant numériquement complexes, le calcul d'une image en séquentiel varie de la seconde à plusieurs minutes suivant la complexité du modèle. L'objectif a été de diminuer ce temps par la parallélisation des algorithmes et l'exécution sur grappes de machines multiprocesseurs afin d'obtenir des animations en temps réel. Différentes méthodes d'intégration des équations du mouvement ont été implantées en parallèle. Dans le cas de l'emploi de méthodes implicites, les opérations coûteuses en calcul proviennent de la résolution de systèmes linéaires par la méthode du Gradient Conjugué impliquant des opérations d'algèbre linéaire de type multiplications de matrices creuses et de vecteurs.

Ce projet de thèse a contribué à l'obtention de nouvelles structures algorithmiques parallèles efficaces avec l'obtention d'algorithmes asynchrones. Il a également permis de valider l'approche de l'environnement de programmation parallèle ATHAPASCAN (projet INRIA-APACHE) avec la mise au point d'applications avec des contraintes temps réel mou ainsi que le contrôle dynamique de son ordonnanceur. Durant ce projet de thèse, un couplage entre la simulation parallèle de textiles et son affichage utilisant l'environnement de visualisation multi-écrans Net Juggler a également été réalisé en faisant communiquer efficacement ces deux programmes parallèles.

Mots-clés Programmation parallèle, simulation de textiles, modèles physiques, couplage de programmes parallèles.

Abstract

Parallel algorithms of physical simulation in computer graphics :
application to cloth simulation

The topic of this Ph. D. thesis combines high performance computing and virtual reality. We propose parallel computing methods for 3D object animation in computer graphics. We target a cloth simulation with physically-based models. The fundamental law of dynamics are used to model with realism the movement of several objects. Due to the numerical complexity of the models used, a frame needs from one second to several minutes of computation on a single CPU. The goal is to decrease this time by parallelizing the algorithms and by executing them on a SMPs cluster to obtain real time animations. Several integrating methods have been parallelized. When implicit methods are used, linear systems are solved with a Conjugate Gradient method involving a high number of computations with linear algebra operations like multiplication of sparse matrices and vectors.

This Ph. D. thesis proposes new efficient parallel algorithmic structures and asynchronous algorithms. This work also validates the approach of the ATHAPASCAN parallel programing environment (INRIA-APACHE project) and its dynamic control scheduler for the implementation of soft real time applications. Code coupling has also been experimented between the parallel cloth simulation and the multi display rendering environment Net Juggler. It enables real time multi display cloth rendering through managed communications.

Keywords Parallel programing, cloth simulation, physically-based models, parallel coupling.