UNIVERSITÉ DE GRENOBLE

Número attribué par la bibliothèque
_/_/_/_/_/_/_/_/_/_/

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **"Informatique"**

préparée au Laboratoire D'Informatique de GRENOBLEdans le cadre de
l'école Doctorale "Mathématiques, Sciences et Technologies de l'Information"

présentée et soutenue publiquement

par

Everton HERMANN

le 30 Juin 2010

Interactive Physical Simulation on Multi-core and Multi-GPU Architectures

---

Directeur de thése : Bruno RAFFIN

JURY

STÉPHANE COTIN	Président
MATTHIAS TESCHNER	Rapporteur
RAYMOND NAMYST	Rapporteur
BRUNO RAFFIN	Directeur de thése
FRANÇOIS FAURE	Co-encadrant

Contents

Chapter 1: Introduction **1**
1.1 Contributions 2
1.2 Organization 4

I Physical Simulation **5**

Chapter 2: Background **7**
2.1 Simulation Pipeline 7
2.2 Physical Models 9
2.2.1 Mass-spring Models 9
2.2.2 Finite Elements 10
2.3 Time Integration 11
2.3.1 Explicit Methods 11
2.3.2 Implicit Methods 11
2.4 Collision 12
2.4.1 Broad Phase Detection 13
2.4.2 Narrow Phase Detection 14
2.4.3 Response 15
2.5 The Sofa Framework 16
2.5.1 Multi-model Representation 16
2.5.2 Scene Graph 17
2.6 Summary 18

Chapter 3: Ray Traced Collision Detection **21**
3.1 Octree Construction 23
3.2 Ray Tracing 23
3.3 Self-collision 25
3.4 Reaction 26
3.5 Evaluation 28
3.6 Collision Detection Parallelization 30
3.7 Summary 33

II Parallel Physical Simulation 35

Chapter 4: Parallel Architectures and Programming 37
4.1 Parallel Architectures . . . 37
4.1.1 Architecture Models . . . 38
4.1.2 Parallel Computer Memory Architectures . . . 39
4.1.3 Specialized Coprocessors . . . 40
4.2 Parallel Programming Environments . . . 42
4.2.1 Message Passing . . . 43
4.2.1.1 PVM and MPI . . . 43
4.2.1.2 Charm++ . . . 44
4.2.2 Shared Memory Parallel Programming . . . 44
4.2.2.1 OpenMP . . . 44
4.2.2.2 Cilk . . . 45
4.2.2.3 Threading Building Blocks . . . 46
4.2.2.4 Athapascan/KAAPI . . . 47
4.2.3 Programming Environments for Heterogeneous Architectures . . . 51
4.2.3.1 Architecture-specify Code Generation . . . 51
4.2.3.2 Scheduling on Heterogeneous Architectures . . . 51
4.3 Previous Works on Athapascan/KAAPI . . . 52
4.3.1 KAAPI Structure . . . 52
4.3.2 Lightweight Thread Execution . . . 53
4.3.3 Dynamic Load Balancing . . . 54
4.3.4 Static Load Balancing . . . 54
4.4 Parallel Physical Simulators . . . 57
4.4.1 Functional Parallelism . . . 57
4.4.2 Collision Groups Parallelism . . . 58
4.4.3 Body Level Parallelism . . . 58
4.4.3.1 Temporal Precedence . . . 59
4.4.3.2 Data Flow Analysis . . . 59
4.4.4 Data Parallel Physical Simulation . . . 60
4.5 Summary . . . 61

Chapter 5: Task-Based Time Integration Parallelism 63
5.1 Sofa Runtime . . . 64
5.1.1 Visitor Level Parallelism . . . 65
5.1.1.1 Component Level Parallelism . . . 65
5.2 Transparent Parallelization . . . 67
5.3 Graph Partitioning and Mapping . . . 68
5.4 Cumulative Write . . . 70
5.5 Dynamic Loops . . . 71
5.6 Partition Stealing and Execution . . . 72
5.7 Results . . . 73
5.7.1 Independent Similar Objects . . . 73
5.7.2 Heterogeneous Scene . . . 76
5.7.3 Domain Decomposition . . . 77

5.7.4 Medical Simulation . . . 79
5.8 Summary . . . 80

Chapter 6: Hybrid Architectures **83**
6.1 Multi-GPU Abstraction Layer . . . 84
6.1.1 Multi-architecture Data Types . . . 84
6.1.2 Transparent Multi-GPU Kernel Launching . . . 86
6.1.3 Architecture Specific Task Implementations . . . 87
6.2 Scheduling on Multi-GPUs . . . 87
6.2.1 Partitioning and Task Mapping . . . 87
6.2.2 Dynamic Load Balancing . . . 88
6.2.3 Harnessing Multiple GPUs and CPUs . . . 89
6.3 Results . . . 90
6.3.1 Colliding Objects on Multiple GPUs . . . 90
6.3.2 Affinity Guided Work Stealing . . . 91
6.3.3 Involving CPUs . . . 92
6.3.4 Evaluating Scene Changes . . . 93
6.4 Runtime Partition Generation . . . 95
6.5 Summary . . . 95

Chapter 7: Conclusion and Perspectives **97**
7.1 Goals . . . 97
7.2 Methodology and Contributions . . . 97
7.3 Perspective Works . . . 99
7.3.1 Locality-aware Collision Detection . . . 99
7.3.2 Reduce Data Transfer in Interactions . . . 99
7.3.3 Dynamic Domain Decomposition . . . 100
7.3.4 Parallel Environment . . . 100

III Résumé en Français **103**

Chapter 8: Introduction **105**
8.1 Contributions . . . 106

Chapter 9: Détection de Collision par Lancer de Rayon pour les Objets Déformables **109**
9.1 Detection . . . 110
9.1.1 Construction de l'Octree . . . 111
9.1.2 Lancer de Rayons . . . 111
9.1.3 Auto-Collision . . . 112
9.2 Réponse aux collision . . . 113
9.3 Résultats . . . 114
9.4 Conclusion . . . 118

Chapter 10: Parallélisation de l'Intégration du Temps en Utilisant des Taches 119
10.1 Generation du Graph de tâches . . . 121
10.2 Partitionnement du Graphe . . . 121
10.3 Boucles Dynamiques . . . 122
10.4 Résultats . . . 124
10.4.1 Objets Identiques Indépendants . . . 124
10.4.2 Scène Complexe . . . 125
10.4.3 Décomposition du domaine . . . 127
10.4.4 Simulation Médicale . . . 127
10.5 Conclusion . . . 128

Chapter 11: Simulation Physique Intéractive sur une architecture Multi-CPU et Multi-GPU 129
11.1 Introduction . . . 129
11.2 Simulation Physique . . . 130
11.3 Couche d'abstraction Multi-GPU . . . 130
11.3.1 Types de données Multi-architecture . . . 130
11.3.2 Lancement de Calcul sur GPU . . . 131
11.3.3 Tâches avec Implémentations Multiples . . . 131
11.4 Ordonnancement Multi-GPU . . . 132
11.4.1 Partitionnement et Affectation des Tâches . . . 132
11.4.2 Équilibrage de Charge Dynamique . . . 132
11.4.3 Harnessing Multiple GPUs and CPUs . . . 133
11.5 Résultats . . . 133
11.5.1 Objets en Collision . . . 133
11.5.2 Vol de Travail Guidé par Affinité . . . 135
11.5.3 Tests avec CPUs et GPUs . . . 135
11.6 Conclusion . . . 136

Chapter 12: Conclusion 137
12.1 Objectifs . . . 137
12.2 Méthodologie et Contributions . . . 137
12.3 Travaux Futurs . . . 139
12.3.1 Détection de Collision Basé sur la Localité des Données . . . 139
12.3.2 Réduire de transfert de données dans les interactions . . . 140
12.3.3 Décomposition de Domaine Dynamique . . . 140
12.3.4 Environnement Parallèle . . . 140

Bibliography 143

List of Figures

2.1 Simulation Pipeline: we start from an initial state (a) to detect collisions between objects (b); using the collision response, external and internal forces, we integrate in time (c); Once the new time step is computed we can visualize it (d); the process is repeated restarting from the collision detection (b), to compute a new time step. Collision processing and time integration can be iterative, as suggested by the arrows. 8
2.2 Mass-spring model: each Degree of Freedom of an object is represented by a mass. The internal force propagation is done by the springs network connecting masses. 10
2.3 Finite Elements Model: the object is subdivided in small regions. The vertices of the elements are associated to the degrees of freedom of the object. 10
2.4 From the initial state the broad phase detects objects in close proximity(a). Using these objects the narrow phase detects colliding primitives (b), then the collision response imposes the reaction needed to keep the objects apart (c). 13
2.5 Proximity collision detection: the primitives (point, edge, polygon) from an object are tested against another object to detect primitives in close proximity 14
2.6 Strong and weak coupling: in weak coupling each object is simulated independently while in strong coupling, colliding objects are simulated by the same solver. 16
2.7 Illustration of the multi-model representation in SOFA. *Left*: possible representations for a simulated object, with the Behavior Model controlling the update of the other representations through a series of mappings. *Right*: examples of these representations for a liver model. Notice how the Visual Model is more detailed than the Behavior Model and how the Collision Model relies on a very different representation. 17
2.8 Left: two interacting bodies. The DOFs are shown as circles, and the forces as lines. A solid line describes an internal force, a dotted line an external force. Right: graph associated to the scene on the left. The nodes of the scene-graph, shown as stars, allow to model structured groups of components. 18

3.1 Collision detection between Object 1(O_1 and O_2) using proximity. Primitives that are farther than the proximity threshold are not detected (green) 22
3.2 Collision detection and response: from a pair of colliding objects (a) we trace rays from the vertex in the intersection zone (b), from these rays we create penalty forces (c) to separate the objects (d) 22
3.3 Quadtree version of the proposed polygon distribution algorithm. (a): a large triangle placed inside the first cell level. (b) and (c): different cases of triangles and their corresponding octree nodes. 24
3.4 Colliding points validation. Point1, point2 and point3 denote the points identified by the Algorithm 3. 25
3.5 Self-collision detection: a ray is traced from the origin point in the direction of the normal; a self collision is found when it hits an outward surface (Intersected point) 26

3.6 Contact force. In (a), a sharp object undergoes a non-null net tangential force. In (b), angle α is used to estimate the quality of the contact model and to weight its force. . . . 27
3.7 A cylinder undergoing various tangential forces due to low geometric resolution. 27
3.8 Ratio of tangential and normal force, against the number of cylinder faces. 28
3.9 Our method can fail in case of non-convex intersection volume (Ray B). 29
3.10 A test scene. Objects starts interpenetrated and we observe the evolution overtime. . . . 30
3.11 Scene used for performance comparison. Objects are not colliding at the beginning. . . . 31
3.12 A deformable chain test. 31
3.13 Performance evaluation with a variable number of colliding tori, each of them including 1600 triangles. 32
3.14 Example of data dependency graph for three colliding objects (A,B and C) when octree and ray traced collision detection are computed in parallel. 33

4.1 Three different levels of parallelization applied to a physical simulator. 38
4.2 Multi-processor architectures, with different memory access models 40
4.3 Architecture comparison between CPU and GPU. On CPU (left), most of the chip area is destined to cache and execution control, while on GPU (right) arithmetic and logic units are predominant. 42
4.4 Sequential Fibonacci algorithm used to compare different parallelization approaches. . . 46
4.5 Fibonacci number computation using Cilk. Each call to spawn creates an asynchronous tasks that can be executed in parallel. The sync keyword is used to wait all the previously launched task and sum their result. 47
4.6 Example of Fibonacci number using Intel TBB 48
4.7 Example of Fibonacci number using Athapascan 49
4.8 Fibonacci execution graph 50
4.9 Hierarchical structure of KAAPI 53
4.10 Steps of static partitioning in KAAPI: From the original program (a) a data flow graph is extracted (b). Each task is mapped to a partition (c). The task are partitioned and the graph enriched with communication tasks (d). Individual stacks of tasks are generated (e). The stacks of tasks (K-Threads) are distributed across the processors (f) 55
4.11 Control tasks employed to guarantee data access coherence between different partitions. The reader waits for the data to be produced by the writer. 56
4.12 Static scheduling execution time. Partitioning enrich the task graph to ensure the communication between partitions. Generation groups the tasks from a partition in a single K-Thread. Deployment: deploys the tasks and data on all the processors 57

5.1 Time integration visitors execution: the Euler Solver Visitors (b) traverse the Scene Graph(c) triggering operation in a sequential order (d) 64
5.2 Parallel physical simulation using a recursive approach and a data flow graph. 66
5.3 Framework Interface 67
5.4 Component specific implementation for the addForce function 67
5.5 Framework side changes to support task level parallelism 68
5.6 Execution flow coupling SOFA and KAAPI. SOFA generates a task graph using the KAAPI interface. This graph is partitioned and execute. During execution the tasks call the SOFA components. 69
5.7 Control tasks employed to guarantee data access coherence between different partitions. Left: initial data-flow graph. Right: graph after partitioning. All the writes are performed in a temporary buffer that will be further accumulated by the Reducer. 70

5.8 Dynamic loop. Functions e1, e2, g1, g2 change multiple objects. Left: standard code with its execution flow. Right: Our modified code with the graph representation of our implementation. 71
5.9 Task deployment structure. 74
5.10 Speedup using 64 identical bars of size 16x4x4 without collision. $T_1 = 150ms$ 75
5.11 Scene used for the complex simulation tests. Objects are simulated using different methods, like Finite Elements, Springs and Regular Grids 76
5.12 Speedup on a heterogeneous scene containing different objects simulated using different mechanical models. 77
5.13 Domain Decomposition Scene: to expose more parallelism a soft tissue is subdivided in horizontal stripes. 78
5.14 Speedup using a mass-spring mesh simulation by domain decomposition. The sequential time for each case is: 125ms (128x128), 500ms (258x258), 2.2s (512x512) 79
5.15 Scene used in medical simulations tests. 80

6.1 Organization of the multi-GPU abstraction layer on top of the run-time environment. . . 84
6.2 Data access on multi-GPU architectures. Only one CPU is attached to a GPU to launch kernels, but any CPU can access a GPU to read or write data. 85
6.3 Multi-implementation task definition. Top: Task Signature. Left: CPU Implementation. Right: GPU Implementation. 86
6.4 Simulation of 64 objects, falling and colliding under gravity. Each object is a deformable body simulated using Finite Element Method (FEM) with 3k particles. 90
6.5 Speedup per iteration when simulating 64 deformable objects falling under gravity (Fig. 6.4) using up to 8 GPUs 91
6.6 Heterogeneous scene results. Objects are simulated using either Finite Element or Mass-Spring model. 92
6.7 (a) A set of flexible bars attached to a wall (a color is associated to each block composing a bar). (b) Performances with blocks of different sizes, using different scheduling strategies 93
6.8 Simulation performances with various combinations of CPUs and GPUs. 93
6.9 Decomposition of a time step. 94
6.10 Decomposition of a time step. 94

9.1 Problèmes avec la détection de collision basé sur la proximité quand les corps se croisent. (a):proximité à l'intérieur et l'extérieur du volume intersection s'annulent. (b): contacts indésirables peuvent être obtenus. Vert: une grande partie de la surface d'intersection est ignoré. 110
9.2 Version quadtree de l'algorithme de distribution de triangle. (A): un grand triangle placé au niveau de la première cellule, (b) 2 triangles stockés au deuxième niveau et (c) 3 triangles au troisième niveau. 111
9.3 Validation des points de collision. Point1, point2 et point3 correspondent aux points identifiés dans l'Algorithme 4. 113
9.4 Détection des auto-collisions. 113
9.5 Force de contact résultante. Image (a), un objet pointu subit à un force nette tangentielle non-nulle. En (b), l'angle α est utilisé pour estimer la qualité du modèle de contact et le poids de sa force. 114
9.6 Un cylindre subit diverses forces tangentielles dues à la discretisation grossiere de l'objet 114
9.7 Ratio de la force tangentielle et normale, par rapport aunombre de faces du cylindre . . . 115
9.8 Une scène de test. Chaque tore est composé de 1600 triangles. 115

9.9 Scène utilisé pour la comparaison des performances. 116
9.10 Test avec une chaîne déformable. 117
9.11 Évaluation de la performance avec un nombre variable de tores en collision, chacun d'entre eux est composé de 1600 triangles. 117

10.1 Simulation Pipeline 119
10.2 Le graphe de taches qui composent un pas de temps de l'integration explicite du temps 5. Gauche: Un approche recursif ajoute des synchronizations apres chaque operation. Droite:nottre aproche permets d'eviter les synchronization en utilisant une analyser de la dependece des donnees. 120
10.3 Tâches de contrôle employées pour garantir la cohérence d'accès aux données entre les différentes partitions. Avec un accès du type lecture/écriture le lecteur attends que les données soient produites par l'écrivain. Dans une mode d'écriture cumulative toutes les écritures sont effectuées dans un tampon temporaire qui seront par la suite accumulés par le réducteur. 122
10.4 Boucle dynamique. Fonctions E1, E2, G1, G2 changent des objets multiples. A gauche: le code standard. Au milleu: notre code modifié. A droite: une représentation du graphe generé avec nottre approche. 123
10.5 (. 124
10.6 Speedup sur une scène hétérogène contenant des objets simulés en utilisant différents modèles mécaniques. 125
10.7 Scene used for the complex simulation tests. Objects are simulated using different methods, like Finite Elements, Springs and Regular Grids 126
10.8 Speedup en utilisant une système de masse-ressort par décomposition de domaine.Le temps séquentiel pour chaque cas est: 125ms (128x128), 500 ms (258x258), 2.2s (512x512)126
10.9 Décomposition de Domaine: pour exposer plus de parallélisme un tissu est subdivisé en bandes horizontales. 127

11.1 Simulation de 64 objets qui tombent avec des collisions. Chaque objet est un corps déformables simulés à l'aide d'un méthode des éléments finis (FEM) avec 3k particules. 130
11.2 Définition d'une tâche multi-implementation. Haut: Signature de la tâche. A gauche: L'implémentation CPU. A droite: l'implémentation GPU. 131
11.3 Simulation de 64 objets avec des collisions. Chaque objet est déformable simulé à l'aide d'une méthode des éléments finis (FEM) avec 3k particules. 134
11.4 Accélération par itération lors de la simulation 64 objets déformables (Fig. 11.3) en utilisant jusqu'à 8 GPUs 134
11.5 (a)Un ensemble de barres souples fixées à un mur (chaque couleur est associée un bloc qui compose une barre). (b) Performances avec des blocs de tailles différentes, en utilisant différentes stratégies d'ordonnancement 135
11.6 Tests de performances avec des diverses combinaisons CPU et GPU. 136

Chapter 1

Introduction

Interactive physical simulation is a key component of realistic virtual environments. However, the amount of computation as well as the code complexity grow quickly with the variety, number and size of the simulated objects. With a complex code, collaboration as well as the development of new algorithms become difficult. Computing intensive simulation, on the other side, can result in poor interactivity.

The Simulation Open Framework Architecture (SOFA) intends to reduce the software complexity by providing a well-defined common interface for physical algorithms. It improves research collaboration, allowing to reuse and easily compare a variety of available methods. The simulated scene complexity is also reduced by combining simple components to generate complex ones. Although the primarily target application of sofa is medical simulation such as virtual surgery, applications from different domains also rely on this framework. Examples includes virtual immersion environments, materials engineering and computer aided design.

During this thesis we studied two ways to obtain a better interactivity from physical simulation. The first one rely on new simulation algorithms to ameliorate the user perception of interactivity even on low refresh rates. The second one takes profit of parallel architectures to speedup the simulation computation, allowing to simulate more complex scenarios in a reduced amount of time.

One way to improve interactivity by changing the simulation algorithm is to use large time steps to bring the virtual time closer to real-time. This approach relies on the way the user perceives the visual result. Even with a low refresh rate the user can comfortably interact with the scene if the time step is large enough to keep simulation time close to real-time. It can be used on integration methods such as implicit Euler with a reduced impact on stability. Unfortunately, other parts of the physical simulation are more sensitive. For instance, when doing collision detection, objects that are apart in a time step can be deeply interpenetrated in the next one. When it happens, collision detection algorithm must be able to separate interpenetrated objects to take advantage of large time steps.

Some applications, such as games, can degrade accuracy to run on slower desktop computers. On the other hand, scientific simulation usually requires huge computing power to be interactive. Taking advantage of parallel architectures like multi-core machines becomes a necessity. However, making the potential power of these architecture accessible to non-experts in parallelism is still a challenge. Individually parallelizing simulation algorithms requires an important effort. In a collaborative work new algorithms are constantly being developed, and often the developers are non-experts in parallelism. To

offer an alternative for non-experts to benefits from parallel machines we intend to exploit the parallelism at a component level, and offer the possibility for developers to write components in a sequential way. The difficult is to identify the independent tasks and at the same time ensure that load is well-balanced across the resources. To provide a transparent parallelization we target on shared memory architectures, avoiding the additional overhead and complexity of distributed memory machines. The complexity of the parallelism aspects is then delegated to the underlying framework.

In addition to shared memory machines with an increasing number of cores, the emergence of machines with many tightly coupled computing units such as graphics cards raises expectations for interactive physics simulations of a complexity that has never been achieved so far. Hundreds of elements can be computed in a single clock using SIMD operations. Current machines capable of delegating computation to graphics cards usually show a mix of standard generic processor cores (CPUs) with specialized ones (GPUs). One particularity of such architectures is that processing units are not only heterogeneous in terms of computing power, but also on the programming model. CPUs are programmed using a Multi-Instruction Multiple Data model, while GPUs are programmed as Single Instruction Multiple Data (SIMD). Communication with specialized processing units is costly and can impair scalability. In fact, specialized units (GPUs) are co-processors usually mounted as an independent card connected over a PCI-Express bus. Transferring data using this PCI bus can have a bandwidth ten times smaller than the device global memory[1].The difficulty is then to efficiently take advantage of all the available parallelism by delegating work to the right device and minimizing the overhead of expensive memory transfers.

1.1 Contributions

When developing a physics simulation engine we can usually decompose it in a pipeline with three main steps: collision detection; time integration and display. Collision detection compares objects geometry to compute if objects are colliding. Time integration makes the objects advance in time based on the physics laws. And display shows the result to the user, which can be visually in a screen, using haptics devices or any other output device. The contributions of this thesis are in the collision detection and time integration phases.

The first contribution is a new collision detection algorithm that is robust when increasing the simulation time step. The quality of the collision detection is much more dependent on the time step than the time integration. Using large time steps, two objects that are far from each other at a given time, can be in a deep interpenetration on the following time step. Continuous collision detection can avoid such problem by interpolating the trajectory of the objects and computing where and when two objects would collide. However, computing continuous collision detection is time-consuming, and in most cases it is not fast enough for interactive simulation. That is why most interactive physics simulators rely on discrete time integration. Unfortunately, traditional approaches, like proximity queries, are unable to solve the cases of deep interpenetrations.

Due to this lack of solutions that are efficient and yet robust on large time steps, we developed a new collision detection and response algorithm. Our discrete time solution is based on ray tracing to detect when two objects are colliding. A ray is shot from each surface vertex in the direction of the inward normal. A collision is detected when the first intersection belongs to an outward surface polygon of another body. A contact force between the vertex and the matching point is then created to separate

[1] Based on tests in a GeForce GTX 280.

the objects. This algorithm has the advantage of computing collision detection and response at the same time, since the same ray used to detect a collision is used to create a contact. Also it is robust to large time step since, since interpenetrating objects can be separated by the created contacts.

To be efficient, our algorithm must be preceded by a filtering phase, usually called broad-phase, that discards collision that can be easily detected as nonexistent. For instance, a broad phase can compare the bodies bounding boxes to determine if bodies are potentially colliding. Our algorithm is then placed in the narrow phase that takes potentially colliding objects, and compares its polygons to compute an accurate collision detection.

With the available multi-core architecture we developed a parallel version of this algorithm to improve collision detection computing time. In our collision detection algorithm, the pairs of colliding objects can be computed independently from each other. We benefit from this inherent parallelism to distribute the pairs over the different processing cores. We then apply a work-stealing load balancing strategy to avoid having idle processors. Experiments in a Quad-core machine has shown that the parallel implementation is more than twice as fast compared to the sequential one. Even if this parallel implementation is targeted at our ray traced algorithm, it can be applied to any other collision detection algorithm that contains a thread safe narrow phase test. This genericity relies on a coarse grain parallelization, as we only consider the parallelism among different pairs of colliding objects.

The approach used to parallelize the collision detection step relies on dynamic scheduling. It fits well to this step first because load balance is difficult to predict and also because most of the tasks are independent and does not requires a deeper evaluation to extract parallelism. Parallelizing the other steps of the simulation pipeline is more difficult since data dependencies are intricated.

Enabling coarse grain task parallelization for the time integration step is another contribution of this thesis. We rely on a hybrid approach that combines static scheduling and dynamic load balancing. When using static scheduling we assume that all the tasks needed to compute an iteration as well as the dependencies between them are known before the execution. We can evaluate the task dependencies to better distribute the work in a way that minimizes communication. Dynamic load balancing on its turn allows to redistributed the load across processors.

Our parallel execution of an iteration on shared memory multi-core machines works as follow. Prior to the execution, we instrument the code, marking tasks and the data shared between them. This instrumented code is used to generate a task dependency graph that contains all the operations needed to perform a simulation time step. Then the graph is partitioned using a static scheduling strategy to enforce data locality and minimize communication between threads. The partitions are distributed over the processors, and a work stealing algorithm is used to redistribute the load over processors.

The choice of a coarse grain task parallelization results in a low impact on physics algorithms, since parallelism is mainly extracted from the coordination code. Additionally, work stealing load balancing provides a seamless multicore level parallelism that does not prevent some physics routines to be accelerated using co-processors. However, the traditional work stealing algorithms dos not deliver satisfying results on hybrid architectures composed by CPUs and GPUs. The memory transfer costs and the difference in the programming model are note taken into account when choosing a task to steal. Our last contribution is to develop new work stealing strategies that takes into account the particularity of this hybrid architecture. Our new stealing criteria are aware of the data locality and the efficiency of the algorithm on each architecture. A high level interface is used to create tasks that have multiple implementations. We can specify for the same task a CPU and a GPU implementation. The choice of the

implementations to be executed is left to our scheduler. It allows a clear separation between the applications code and the parallel environment one. Experiments using 4 GPUs and 4 CPUs show that we can make CPUs and GPUs cooperate to obtain better speedups than what is obtained when summing the speedups obtained individually by each architecture.

1.2 Organization

This thesis is decomposed in two parts. The first one contains the contributions on physical simulation. We start, in Chapter 2 with an overview of the physics simulation algorithms, to help understanding the reminder of this thesis. Then in Chapter 3 we present our ray traced collision detection algorithm, together with the collision detection parallelization using coarse grain tasks.

The second part groups the contributions on parallelism. First we review some basic concepts of parallelism: the programming model and classification of parallel architectures. Then we present the related works and the different ways to use parallelism on physics simulation. In Chapter 5 we show the multi-core parallelization of the time integration step of physics simulations. This approach uses task parallelism which combines static scheduling to detect parallel task, and dynamic scheduling for load balancing. It is targeted on multi-core architectures composed by general purpose processors. In Chapter 6 we extend our works to hybrid architectures composed by CPUs and GPUs by changing the way load balancing is done.

Part I

Physical Simulation

Chapter 2

Background

The goal of physics simulation is to reproduce the dynamics of objects submitted to mechanics laws. These objects can have specific properties like rigid bodies, deformable bodies and fluids, i.e. they can be simulated using different algorithms. Together with these physics methods we also have collision detection and response algorithms, which create interactions between objects, for instance by creating forces to separate colliding objects. Finally we have the ordinary differential equation solver (ODE solver) that is in charge of computing the new time-step values.

All these components are combined to create a simulation. Computing such a system is complex and computationally expensive. The overall computation time of each object depends on many factors, like the geometric resolution, the complexity of the behavior models, the ODE solver, the time step size, as well as the desired precision. Usually the more accurate method is also the more time-consuming one. Because of this trade-off between accuracy and computing time, simulating complex scenes at an interactive refresh rate is a challenge for many applications like video-games, virtual training or virtual surgery. Finding the method that best fits to desired result is determinant to obtain a good balance between accuracy and computing time.

In the remainder of this chapter we will explain how these components: the physical models, collision detection, collision response, etc. can be combined to create a physical simulation. We will start by showing the phases of a common simulation pipeline followed by a description of the most popular physical models. In Section 2.3 we will present the algorithms that can be employed to advance the objects in time. Finally some methods employed in collision detection and response are presented in Section 2.4.

2.1 Simulation Pipeline

The physical **simulation pipeline** is an iterative process where a sequence of steps is executed to advance the scene forward in time. We call **scene**, the data structure used to describe the objects that we want to simulate. Usually it contains the objects state, like position and velocity and the properties of the objects material. In SOFA this scene is represented by a graph that contains a hierarchy of objects together with the algorithm used to simulate their behavior (see Section 2.5.2).

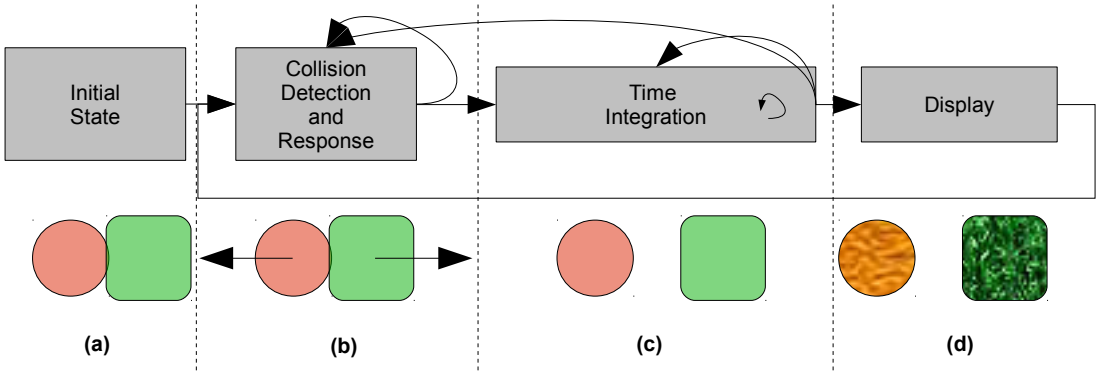


Figure 2.1: Simulation Pipeline: we start from an initial state (a) to detect collisions between objects (b); using the collision response, external and internal forces, we integrate in time (c); Once the new time step is computed we can visualize it (d); the process is repeated restarting from the collision detection (b), to compute a new time step. Collision processing and time integration can be iterative, as suggested by the arrows.

In most cases a simulation pipeline is composed by three steps: collision detection; time integration and visualization, as shown in Figure 2.1. Each one of these steps is usually subdivided in smaller tasks. For example the collision detection step contains its own internal pipeline. Here we consider that collision detection takes place before time integration, but collision processing and time integration cant take place in different orders, depending on the specific methods.

In the collision detection step, the geometric primitives of each object are tested against each other so that we can decide when two objects are colliding or not. To react to a collision, the simulator can dynamically create or delete interactions between objects based on geometry intersection found during the collision detection. These interactions will influence the time integration computations in the attempt to cancel the intersections at the end of the time step.

Time integration consists in computing a new state (i.e. position and velocity vectors), starting from the current state. Giving a time interval it integrates the forces in time, using this value to compute the new velocity and position. Stable methods usually involve solving a linear equation system to determine the new state of the objects, and depending on the method employed to solve the equation, this step can contain an iterative phase, for example using a conjugate gradient.

Finally, after computing the new state the result is delivered to the visualization step. Usually the objects are rendered and displayed to the user. In some scientific simulation it can be just a set of values displayed to the user or saved to a file. In other cases the result of the time integration step can also be sent to an external tool like haptic devices. Although the visualization step is as important as any other phase of the simulation pipeline it is not in the scope of our research work. Our works focus on **collision detection** and **time integration** steps.

2.2 Physical Models

The phases of a simulation pipeline represent the high level operations needed to compute a time step. Internally each phase executes the specific algorithms for each type of physical model. When launched by the simulation pipeline, these specific algorithms update the time-dependent variables of the object.

The internal data representation of dynamic bodies is directly related to their allowed movements, called Degree of Freedom (DOF). The set of DOFs completely specifies the displaced or deformed position and orientation of the body. For instances, a rigid body in a three-dimensional environment has six independent degrees of freedom, three for the displacement, and three to represent the rotation. A deformable object can be represented by a set of particles, each one containing three degrees of freedom: one for each coordinate in the space. In other words, such a deformable object with N particles will have $3N$ degrees of freedom.

The time-dependent variables that are updated during the simulation are mainly the position and velocity of each Degree of Freedom (DOF). Other data are stored internally and are specific to a physical model. For instance, it can contain the links between the particles, the material parameters, etc. The physical model algorithm uses these internal data to control the way the force acts on each DOF and also how the forces are propagated inside an object. In the following sections we will detail the Mass-Spring System, and the Finite Elements Method, two of the most popular models used in physical simulation. Other algorithms used on physics simulation includes Free Form Deformation [115], Smoothed-particle Hydrodynamics [93], Leonard-Jones Molecular dynamics [76], Rigid Bodies [23].

2.2.1 Mass-spring Models

The Mass-Spring system is one of the simplest physically based deformation model. This method is widely used for modeling soft bodies, specially cloth simulation [69][126]. It is easy to implement, highly parallelizable [57], and fits the needs of simple virtual reality applications. As the name suggests, in this model an object is represented as a set of masses connected by corresponding springs in a fixed topology like shown in Figure 2.2.

At a given simulation time, the state of a mass-spring object can be defined as the position and velocity of each DOF, here represented by masses. The internal forces acting on each mass is computed from the potential energy of each springs connected to the neighbors masses. Often the springs follow a linear force deformation law, although nonlinear springs can be used to model inelastic behaviors.

The simplicity of this method is the reason of its success, as well as its weakness. A spring can only represent one-dimensional deformations, which is not well adapted to represent the structure of three-dimensional objects. Volume conservation is also difficult to achieve. Some works try to improve the fidelity of a mass spring, resulting on visually plausible simulations [68, 15, 34]. However it is hard to obtain a result that is valid to continuum mechanics as there is no underlying reference physical model to find what values must be used as parameters to the system to respect the material properties (stiffness, density, etc).

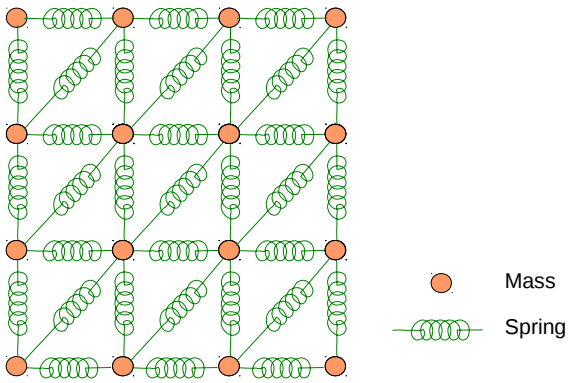


Figure 2.2: Mass-spring model: each Degree of Freedom of an object is represented by a mass. The internal force propagation is done by the springs network connecting masses.

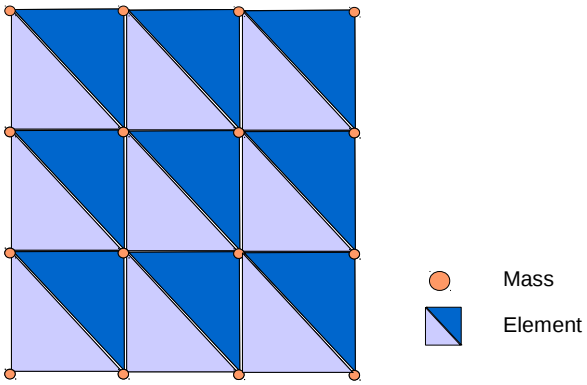


Figure 2.3: Finite Elements Model: the object is subdivided in small regions. The vertices of the elements are associated to the degrees of freedom of the object.

2.2.2 Finite Elements

Unlike mass-spring that is a discrete model, the Finite Element Method [108, 99] is derived from the equations of continuum mechanics. One advantage of having a method that is a direct representation of mechanics is that we can quantify the precision of such a simulation, since the parameters of the model can be taken from experiments on real world objects. For this reason it is one of the most used methods for scientific physics simulation [96, 41, 95, 98].

The main idea of continuum based deformable modeling consists in the minimization of the stored deformation energy all over the volume. The object reaches equilibrium when its potential energy is at a minimum. In FEM the continuous model is discretized by dividing the object into small interconnected regions, called finite elements as shown in Figure 2.2.2. The constitutive laws are approximated by interpolation functions associated with each element.

2.3 Time Integration

The time integration algorithm is the engine of a physical simulation. It is responsible for integrating the motion equations of the interacting objects and particles and compute their trajectory over the time. This system evolution is computed numerically by solving an ordinary differential equation system, to compute the new position and velocity based on the mass and the acceleration.

$f(x, v) = M.a$

There are essentially two groups of integration methods: the explicit methods (Explicit Euler, Runge-Kutta, Explicit midpoint) and their corresponding implicit methods (Implicit Euler, Implicit Runge-Kutta, Implicit midpoint). In the next sections we will explain some of these most common numerical integration methods.

2.3.1 Explicit Methods

The explicit methods compute the state for the next time step by extrapolating the previous state. The first and the second derivatives of the current state are employed to directly create the new state. For example in the Explicit Euler method we assume that the derivatives are the same over the entire time steps and equal to their values at the beginning. It means that the velocity is considered constant during the whole step.

In Algorithm 1 we show the operations needed to perform a time step using an Explicit Euler solver for time integration. The velocity and position derivatives are computed directly from the acceleration and velocity from the previous step. This simple approach has the advantage of being easy to compute due to the reduced number of operations needed to integrate in time.

Algorithm 1 Simulation Pipeline with Explicit Time Integration

$f_t =$ computeForces()
$a = \frac{f_t}{M}$
$x_{(t+\Delta t)} = x_0 + v_0 \Delta t$
$v_{(t+\Delta t)} = v_0 + a\Delta t$

The main limitation of this method is the time step size. There is a critical stiffness value above which the numerical resolution of the system is divergent, called *Courant Condition*. The convergence time step that satisfies the *Courant Condition* is inversely proportional to the square root of the stiffness. It means that when we have a stiff object we will be forced to use a small time step. Such restriction is sometimes prohibitive for real-time physical simulation. That is why most of the work in real-time simulation using explicit integration involves soft bodies [43, 42].

2.3.2 Implicit Methods

The implicit integration methods compute the state by solving an equation involving the current system state and the state at the end of the time step. It means that differently from the explicit method where we employ the derivatives from the values at the beginning of the step (t_0), in an implicit the first

and second derivatives at the end of the step ($t_0 + \Delta t$) are employed to update the system state.

If we rewrite the acceleration equation using an implicit method we obtain:

$a = \frac{f(t+\Delta t)}{M}$

We can approximate the force at $t + \Delta t$ using Taylor series:

$f_{t+\Delta t} = f(t) + \frac{\partial f}{\partial x}\delta x + \frac{\partial f}{\partial v}\delta v$

After applying this force to Newton's second law and developing the equation as presented by [25] we obtain an equation system that can be solved iteratively to compute the acceleration (a):

$(M - \delta t\frac{\partial f}{\partial v} - \delta t^2\frac{\partial f}{\partial x})a = f(t) + \delta t\frac{\partial f}{\partial x}v_0$

Algorithm 2 Simulation Pipeline with Implicit Euler Time Integration

$(f_t, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial v})$ = computeForces(x,y)
solve $\left((M - \delta t\frac{\partial f}{\partial v} - \delta t^2\frac{\partial f}{\partial x})a = f(t) + \delta t\frac{\partial f}{\partial x}v_0)\right)$
$v_{(t+\Delta t)} = v_0 + a\Delta t$
$x_{(t+\Delta t)} = x_0 + v_{(t+\Delta t)}\Delta t$

As seen in Algorithm 2, using an implicit method requires solving an equation to compute the derivatives at $t_0 + \Delta t$. Since it may be nonlinear, solving such equation requires an iterative solution method. The extra computation required to compute an implicit time integration, and the complexity of developing an efficient equation solver can encourage the usage of explicit methods. However, many of the real life problems involve stiff objects, for which an explicit method imposes small time step to avoid numerical instability. For this kind of problem it is preferred to use an implicit method with larger time steps, than using an explicit method with small time steps. This unconditional stability of implicit methods is the reason of their good acceptability in real-time simulation [25, 65, 127, 44].

On the other hand it does not mean that an implicit method will always be better than an explicit one. Whether one should use an explicit or implicit method depends on the problem to be solved. One drawback of the implicit methods is the underestimation of the particles displacement, resulting on smoothed movements. Depending on the approximation criteria, it can leads to some viscosity artifacts.

2.4 Collision

Collision detection is one of the major computational tasks in physically based animation, specially when considering deformable objects. Interactive applications such as cloth simulation, surgery simulation, or even some classes of games, requires efficient collision detection algorithms for deformable bodies. Several works have been done for this class of collision detection algorithms, and a good survey on this domain is presented in [122].

The complexity of a collision detection relies on the geometrical shape of the objects. While special shapes such as spheres or cubes allow the use of optimized methods, the general case of objects bounded by triangular meshes is much more complex.

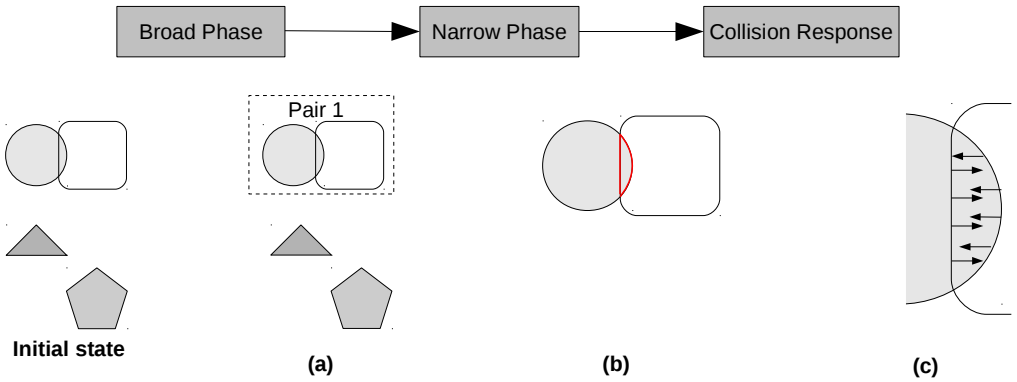


Figure 2.4: From the initial state the broad phase detects objects in close proximity(a). Using these objects the narrow phase detects colliding primitives (b), then the collision response imposes the reaction needed to keep the objects apart (c).

Considering n as the number of objects in the scene, the collision detection requires testing $O(n^2)$ pairs of objects. To test a pair of objects, if there is no acceleration structure, one must compare every primitive of an objects against the primitives from the other. To reduce this computational cost, collision detection can be decomposed in a **broad phase** and a **narrow phase**, like presented by [38]. The broad phase detects pairs of objects that are in close proximity, and excludes pairs that are obviously not colliding. These pairs of objects are then tested by the narrow phase that will exactly determine if the objects are colliding.

The result of the narrow phase is a set of pairs of primitives that are in contact. A collision response strategy is then applied to these points. If we consider the three operations we just described we can represent the collision detection process as a pipeline composed by the **broad phase**, the **narrow phase** and the **collision response** as shown in Figure 2.4 and discussed in the following sections.

2.4.1 Broad Phase Detection

The Broad Phase is used to eliminate pairs of objects that are easily identifiable as non colliding, such as objects that are distant in the three dimensional space. Instead of performing tests on the objects primitives, the collision tests are done using simplified representation of the object such as Bounding Volumes so that testing two object is very fast. A simple approach that is widely used is the **axes-aligned bounding boxes** method.

Structures like a hierarchy of bounding volumes [125] can be employed to obtain a more accurate filtering of potentially colliding pairs. To have a better fitting of the objects some works propose using different bounding shapes, like oriented bounding box (OBB) [58] or discrete oriented polytopes (DOP) [82]. All these structures are lightweight, and at the same time offer an efficient way to reduce the number of collision pairs on the narrow phase.

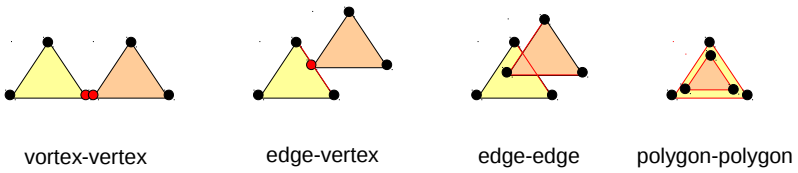


Figure 2.5: Proximity collision detection: the primitives (point, edge, polygon) from an object are tested against another object to detect primitives in close proximity

2.4.2 Narrow Phase Detection

The pairs of objects not filtered off by the broad phase are processed by the narrow phase. More refined and accurate algorithms to detect intersections are employed by the narrow phase. It usually involves comparing geometrical primitives to compute where they are colliding or not. Due to the complexity and the large number of primitives usually employed to represent the objects, the narrow phase tends to be much more compute-intensive than the broad phase.

For rigid objects, the most efficient approaches rely on signed **distance fields** [52]. The distance from an object in a scene is computed and mapped to a Cartesian grid as part of preprocessing. With a distance grid we can compute rapidly the distance of a point to an object without testing the geometric primitives. Each point of one object (which we call the *colliding* object) is tested against the distance field of the other (called the *collided* object). If the point is inside the collided object, the nearest point on the collided surface is found and a constraint between these points is created. The test can also be performed the other way round, by switching the colliding and collided objects. Computing a distance field is a compute-intensive task that is performed once at initialization time for rigid objects, and defined with respect to a local reference frame.

When applying this method to deformable objects, the distance field must be recomputed at each time step. The cost of recomputing this distance field makes this approach too complex for real-time applications. Even if this approach is used on deformable bodies [49], it is not fast enough for interactive simulation. When used on deformable bodies like in [53] it requires a rigid *collided* object.

The most popular strategy is thus to detect pairs of geometric primitives in close **proximity**, setting up constraints to keep them apart. In this approach, the contact points are the ones having a distance to a geometric primitive of the collided object that is below a given arbitrary proximity threshold. This proximity test can be done using different types of primitives pairs, like vertex-polygon, edge-edge, vertex-edge and vertex-vertex like shown in Figure 2.5. In this method a collision is represented by a pair of elements, one from each colliding surface region, and a distance, which represents the distance of the two concerned elements.

However, on discrete time integration the objects can interpenetrate each other from a time step to the other. Since the proximity algorithm compares the distance of two primitives to determine if they are colliding, when primitives go deeper than the proximity threshold they cannot be identified as a contact points (Figure 3.1). Missing these contact points can result in a wrong collision response, and in most cases will not separate the objects.

The problem of surface crossing due to discrete time integration can somehow be alleviated using sophisticated strategies based on **collision prediction**, given current positions and velocities. However, these methods are complex. Their convergence is unclear and they may require short time steps, while large time steps are preferable for real-time applications. Consequently, they have been mainly applied to off-line cloth simulations.

For volumetric objects a reasonable amount of intersection can be visually acceptable and a robust contact modeling method would authorize large time steps. Most of the GPU-based methods can detect the pixels of a colliding surface inside a collided body, but they do not compute the matching points on the collided surface. This prevents a robust setting of the associated contact constraints. In a recent work [48] an extension of the Layered Depth Images (LDI) was proposed to model the intersection volume of the objects, and minimize the intersection between them. Together with the intersection volume, it models the derivatives with respect to the vertex coordinates, allowing to straightforwardly derive the contact forces applied to the vertices.

2.4.3 Response

As seen in Section 2.3 the DOF are updated by solving ordinary differential equations to compute the new object state. Interactive mechanical simulation involves multiple objects of different kinds in the same scene and the simulation evolution can make objects collide. The results of the collision detection algorithm must be considered in the integration scheme. Contact response is generally based on two approaches: constraint-based algorithms [24, 91, 105] and penalty forces [25]. Constraint-based methods use Lagrange multipliers to completely avoid intersection. Penalty-based methods apply repulsion forces which do not always avoid intersection but results in equation system that are easier to solve. In this work we focus on penalty-based methods, the default method in SOFA.

In [10] the objects are simulated independently using their own encapsulated simulation methods. Interaction forces are periodically updated based on the current states of the objects. Aside the interaction force there is no other information from the external state of the object. This approach provides a high flexibility since arbitrary objects can be combined. However, it is limited to explicit time integration, where each object evaluates its net force at the current time to straightforwardly derive its next position and velocity. Interaction forces cannot be taken int account in implicit integration, because the interaction forces are considered constant during each time-step. The objects cannot anticipate the variations of the interaction forces, since these forces depend on several objects. This is sometimes called *weak coupling*. Consequently, divergence occurs unless sufficiently small time steps are applied, which can result in very slow simulations when stiff interaction forces are applied. Unfortunately, high stiffness is generally required for contact forces to avoid visible object intersections.

This well-known stability problem can be avoided using *strong coupling* such as implicit time integration [25], where force variations are anticipated, allowing large time steps and high performance. However, this requires to set up and solve an equation system involving the objects and their interaction forces, which is not possible when the objects are simulated independently.

Using *strong coupling*, the sets of objects are updated as the collision detection pipeline creates and removes contacts based on geometry intersections. It means that objects are grouped following their collision state. As shown in Figure 2.6 objects in the same group are solved by the same solver, and each of these groups can be processed independently from each other.

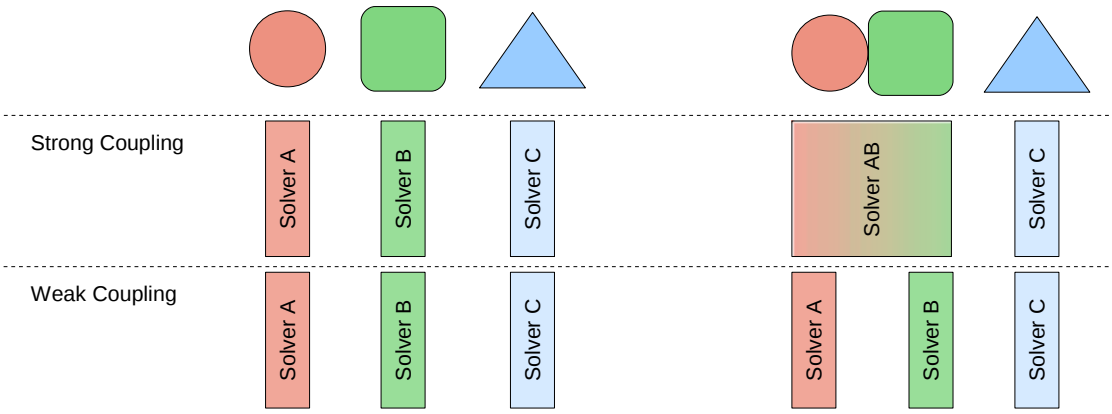


Figure 2.6: Strong and weak coupling: in weak coupling each object is simulated independently while in strong coupling, colliding objects are simulated by the same solver.

2.5 The Sofa Framework

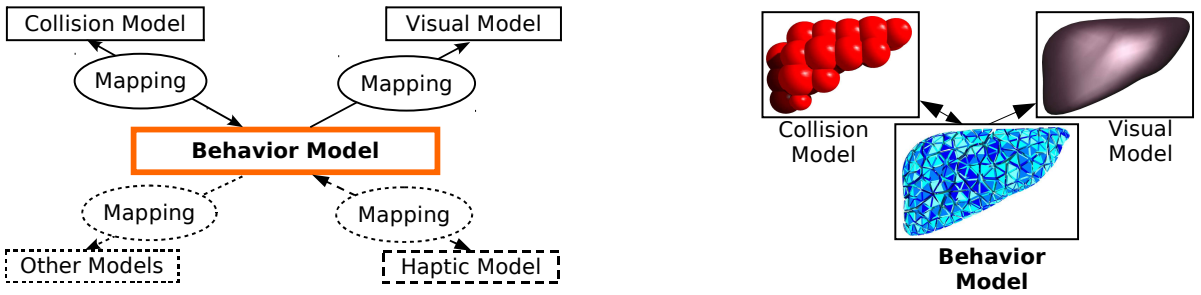
SOFA [11] is an open source framework primarily targeted at medical simulation research. It gathers independently developed algorithms to interact together within a common simulation. Sofa aims to promote collaboration among research groups by providing a common software framework for the medical simulation community. With a common framework it becomes easier to share components, validate and compare new algorithms.

To achieve these objectives, SOFA relies on two main concepts; multi-model representation and a scene graph structure, described below.

2.5.1 Multi-model Representation

In the previous sections we presented the different algorithms employed by each step of a simulation pipeline. The level of detail used to represent an object has an important influence on the resulting performance. In most cases the best balance between performance and accuracy is obtained using different object representations for each algorithm. For example, we can have a coarse mesh employed to the mechanics simulation and use a finer mesh for collision detection and obtain more accurate collision tests. The visualization could also have a finer mesh to produce visually appealing results. This kind of scenario requires multiple models to represent an object behavior. A relationship between them must be obtained to ensure a coherent representation of the bodies all over the simulation.

In SOFA an object is explicitly decomposed into various representations, in such a way that each representation is more suited toward a particular task – rendering, deformation, or collision detection. Then, these representations are linked together so they can be coherently updated. The link between these representations is called *mapping*. Various mapping functions can be defined, and each mapping will associate a set of primitives of a representation to a set of primitives in the other representation (see Figure 2.7). For instance, a mapping can connect degrees of freedom in a Behavior Model to vertices in a Visual Model.



(a) Representation for a simulated object

(b) Example for a liver model

Figure 2.7: Illustration of the multi-model representation in SOFA. *Left*: possible representations for a simulated object, with the Behavior Model controlling the update of the other representations through a series of mappings. *Right*: examples of these representations for a liver model. Notice how the Visual Model is more detailed than the Behavior Model and how the Collision Model relies on a very different representation.

The simplest case is when the DOFs from a model match exactly the DOFs from the other. However, if the geometry differs, a more complex transformation must be employed. One can use barycentric coordinates to update the position of models based on the DOFs from another one.

2.5.2 Scene Graph

Another key aspect of SOFA is the use of a scene graph to organize and process the elements of a simulation. Each component of the scene is attached to a node of a tree structure, allowing to construct a hierarchical representation of the bodies. This hierarchical structure can be used, for instance, to group components representing a same simulated body. The scene-graph can also be dynamically reorganized, allowing for instance the creation of groups of interacting objects. Such groups can then be processed as a unique system of equations by the solver, thus permitting to efficiently handle stiff contact forces (cf. strong coupling in Section 2.4.3). Another advantage of using a scene-graph is that most computations performed in the simulation loop can be expressed as a traversal of the scene-graph. This this component that traverse the graph is called a *visitor* in SOFA. For instance, at each time step, the simulation state is updated by sending an `Animate` visitor to all `Solver` components. Each `Solver` then forwards requests to the appropriate components by recursively traversing its sub-tree using its own visitor.

To illustrate the modularity in SOFA and the use of a scene-graph, we consider the example illustrated in Figure 2.8. In this example, two simulated objects, a rigid square and a simple Mass-Spring model, move through space and eventually collide. To compute the motion and deformation of the objects, we need to define for each of them a set of DOFs and a set of internal and external forces. The DOFs component of the mass-spring model corresponds to the mass-points, while for the rigid object corresponds to the position and orientation of the center of mass. This implies different data types for the DOFs of each object; a set of 3D vectors for the mass-spring and a 3D vector with a quaternion for the rigid object. Contacts between objects are possible through Collision Models associated with each object. The Collision Model for the mass-spring object consists of a set of vertices coincident with the DOFs of the object. The Collision Model for the rigid object, the square shape in Figure 2.8, is rigidly attached to the body reference frame through a mapping. The mapping component is responsible for

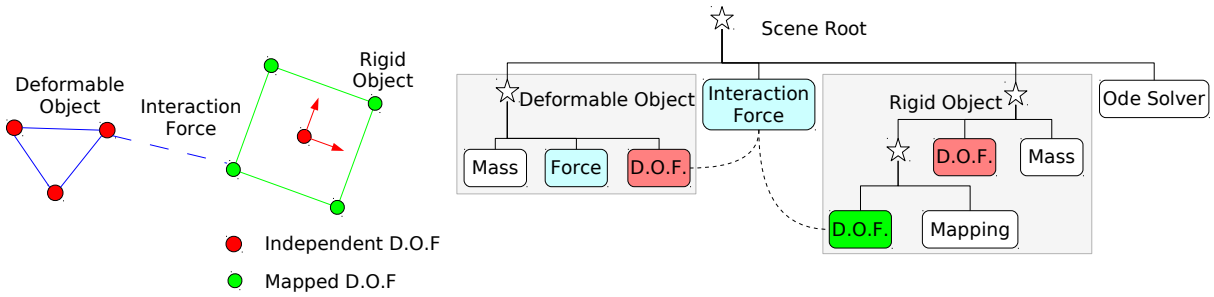


Figure 2.8: Left: two interacting bodies. The DOFs are shown as circles, and the forces as lines. A solid line describes an internal force, a dotted line an external force. Right: graph associated to the scene on the left. The nodes of the scene-graph, shown as stars, allow to model structured groups of components.

propagating the motion of the rigid body to the vertices of the Collision Model, and when collision occurs, the contact forces applied to the Collision Model are propagated back to the DOFs of the rigid body object. Since the vertices of the Collision Model do not coincide with the DOFs of the rigid object, we attach them to a different node of the scene-graph. However, as their motion is totally defined by the rigid body, they are not independent so this new node is created as a child of the rigid body node. The interaction force acts on the collision model vertices, independently of whether they are actual or mapped DOFs. At this point, visitors can be propagated through the scene-graph to simulate both objects as a combined mechanical system.

The usage of scene-graph and visitors have the advantage of clearly separating the component implementations from the high level algorithms. Each component implements the entry points for the operations described by the framework, for instance, compute forces, compute force derivatives, etc. The scene graph then describes how these components are combined, for instance using Finite Elements with proximity-based collision detection, simulated using an Euler Solver. The visitors are then charged to orchestrate the execution and trigger the operations on each component. We can execute these operations in parallel by changing the way the operations are triggered in the components. This parallel execution will be discussed in Part II.

2.6 Summary

The most used algorithms in interactive simulation are those that are simple, efficient and robust for a given scenario. For example mass-springs is usually employed on cloth simulation while Finite Elements is largely used when we need to set parameters reflecting real life objects. On collision detection, distance fields are the most used algorithm when dealing with rigid bodies. When considering deformable bodies, we prefer algorithms like proximity based collision detection, that does not rely on huge precomputing. The choice of the time integration algorithm also depends on the target scene. Explicit methods are simpler but their usage on stiff objects can lead to instabilities, on the other hand implicit methods are harder to implement and time expensive, but their result is unconditionally stable.

In an interactive context, these algorithms must execute at frame rates that provides a reactive response to user actions. Using large time steps usually improves the simulation interactivity. It counterbalances a slower simulation by reducing the gap between the virtual time and the real world time. Another way to improve the interactivity is to speedup the execution of an iteration.

Some algorithms presented in this chapter are not robust to large time steps. For instance, some collision detection algorithms. The object position can radically change between time steps, which can result in deep objects interpenetrations. Collision detection algorithms must avoid such an interpenetration to support large time steps. To better address this aspect of interactive physical simulation we will present in the next chapter a collision detection algorithm that is robust to large time steps.

Chapter 3

Ray Traced Collision Detection

The algorithm presented in this Chapter was published in [66] and is a part of the collision detection module of SOFA [5].

As explained in the previous section, in discrete time integration the surfaces can cross each other from a time step to the other. If this interpenetration is deeper than the proximity threshold these primitives cannot be identified as contact primitives. It results in poor collision responses that sometimes leave the bodies in an interpenetrating state.

Using large time steps increases the probability of having interpenetrating objects. However, for interactive simulation we are forced to use larger time steps to match the virtual time with the real-time. To improve the collision detection robustness we have developed a new collision detection and response algorithm so that interpenetrated objects can be separated during the collision response even in cases of deep intersection.

Our algorithm works with pairs of potentially colliding objects. Considering a collision pipeline, like presented in Section 2.4, our algorithm acts at the Narrow Phase level. It requires a previous broad phase to identify pairs of objects in close proximity. Given such a pair of objects, it finds pairs of colliding primitives. Collision response forces are then applied to these pairs of primitives. Differently from other approaches where collision response is computed separately from the collision detection, by using our algorithm we can obtain the penalty forces together with the collision detection.

To identify these pairs of colliding primitives, we take a vertex on an object surface and follow the opposite direction of the normal up to finding a primitive in the other object. Our approach solves collisions even when objects are deeply interpenetrated and polygons are not close enough to be detected using proximity (Figure 3.1).

The search path starting from a vertex in the surface of the colliding object can be defined by a ray whose origin is at the colliding object vertex and the direction is the opposite of the normal of the vertex at the origin. Once two objects are interpenetrated and colliding primitives are detected, the collision reaction forces are applied to separate them, as shown in Figure 3.2. This penalty force is applied to each object at the point crossed by the traced ray.

This method does not rely on precomputed data to determine the colliding points. Therefore, it is

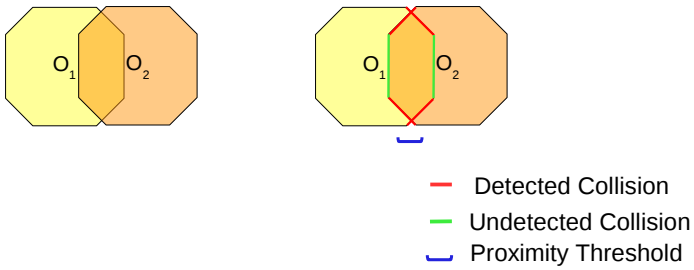


Figure 3.1: Collision detection between Object 1(O_1 and O_2) using proximity. Primitives that are farther than the proximity threshold are not detected (green)

well suited to deformable objects, where the distance field is too expensive to be recomputed at each time step.

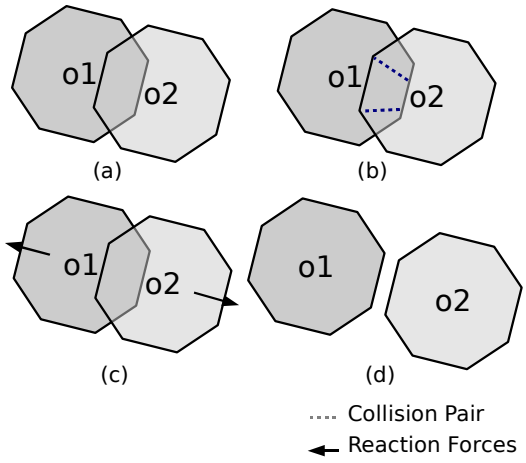


Figure 3.2: Collision detection and response: from a pair of colliding objects (a) we trace rays from the vertex in the intersection zone (b), from these rays we create penalty forces (c) to separate the objects (d)

A naive implementation of ray tracing is to compare the ray against every triangle in an object. However, such approach is prohibitively slow. We accelerate this computation using an octree. For each colliding object we store all the polygons in an octree. Therefore we can easily navigate inside this octree and efficiently find the points crossing the ray.

In the next sections we detail how this octree is constructed, and how we use this structure to find pairs of colliding points between two objects.

3.1 Octree Construction

Finding ray-object intersection is computationally expensive. Without acceleration, each ray has to be tested with all the primitives of an object. Among the most used structures to accelerate ray tracing we have uniform grid, octrees, Kd-trees and BSPs [16].

The acceleration structure employed in our algorithm is an octree, since large empty zones can be easily discarded. Constructing an octree is straightforward as we recursively subdivide a cell in eight sub-cells of the same structure and size, unlike is done by kd-trees or BSP-trees. It is important to have a structure that can be quickly built when considering deformable objects, as we need to update this octree at each time step.

Although using an octree can sometimes result in less efficient queries due to the uniform subdivision of the space, it is easier to compute a ray trajectory inside an octree. The only information needed to compute the next visiting cell are the coordinates of the current cell and the ray. It results in a simple and efficient traversing algorithms since we do not need to deal with cells of different structures and shapes.

For each potentially colliding object we create an octree with the object polygons. Using this spatial data structure we can easily find the polygons that intersect an arbitrary region. The efficiency of the octree depends on the polygon spatial distribution. One way to construct the octree is to successively refine the octree, by splitting the cell while it contains more than one polygon. At the end of this process we will have an octree where each cell has exactly one polygon. On the other hand we have no control on the number of cells a polygon belongs to. In some cases a large polygon will be represented by several small cells, specially in the border between two polygons.

Our approach intends to avoid storing a polygon in a large number of octree cells. We want to have a cell size that is proportional to the polygon size. Larger polygons will be stored in larger cells, and smaller polygons are stored in deeper levels 3.3. We consider the largest dimension of the polygon's bounding box and split down our octree cells to the smallest size that can still fit this bounding box. The depth level (d) of polygon (P) in an octree (O) can be computed by:

$$d = \log_2 \frac{O_{size}}{P_{size}}$$

Figure 3.3 shows a 2D representation of the algorithm employed to construct the octree. This algorithm ensures that each polygon is present in at most eight octree nodes, giving a good equilibrium between octree precision and the number of cells to be tested when traversing the octree. This kind of approach works like a hash giving a fast way to reach a subset of primitives, like done by [121].

3.2 Ray Tracing

We trace a ray starting from vertices located in the zone represented by intersection of the object bounding boxes, which allows us to cull out numerous tests. Our algorithm is decomposed in two phases (see Algorithm 3) : search for colliding pairs, and result filtering.

The search phase consists in taking the opposite of the point normal, and following this direction to

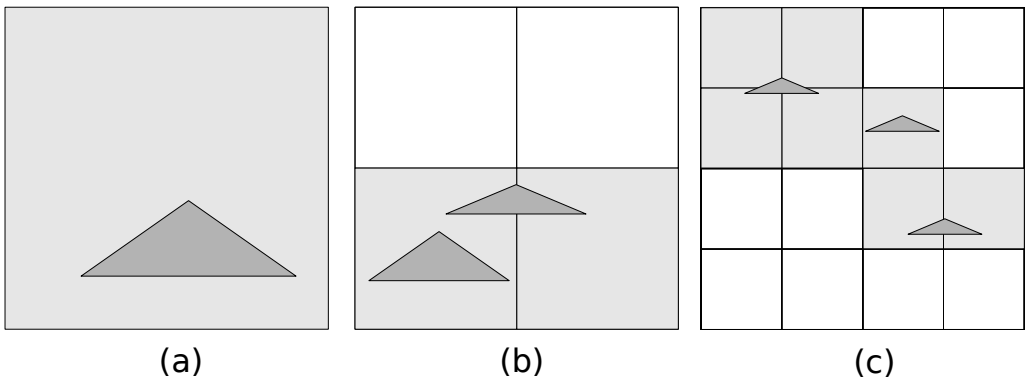


Figure 3.3: Quadtree version of the proposed polygon distribution algorithm. (a): a large triangle placed inside the first cell level. (b) and (c): different cases of triangles and their corresponding octree nodes.

find a point on the other object. The octree cells are visited using the octree traversal algorithm presented by [111]. Each cell of the octree contains a list of polygons that intersect this cell. When a cell is visited, all the polygons it contains are tested against the ray using the algorithm from [92]. If an intersection point is found, this algorithm gives us its coordinates and the distance from the ray's origin. To avoid testing twice the same primitive, we maintain a list of the primitives already tested at the current level.

Algorithm 3 Collision detection Algorithm

Require: $Object1, Object2$
Ensure: pairs of colliding points between Object1 and Object2
  for each point1 in Object1 **do**
    $point2$=traceRay(point1, - point1.normal, Object2)
    if angle between point1.normal and point2.normal $\leq 90^\circ$ **then**
      continue with the next point
    else
      $point3$=traceRay(point1, - point1.normal, Object1)
      if distance(point1,point2) $\leq$ distance(point1,point3) **then**
        add collision pair to the collision response
      end if
    end if
  end for

Once we have a pair of colliding points, one on each object, we test the validity of the resulting contacts. The first verification concerns the angle between the normals of both points. An acute angle means that the ray is entering the second, while an obtuse angle means that the ray is exiting. We are interested on rays that hits an outward face of the object. Hitting a face in the outward direction means that the ray was traced from inside the object, which means that a collision is happening. Eliminating acute angles avoid the misdetection of collision pairs like the one shown in Figure 3.4(a), where two different rays are traced from O_1, but only one is valid, as the angle A is acute. Applying forces to these invalid points would make the objects penetrate even more.

However, only discarding rays that traverse the second object from the outside face is not enough.

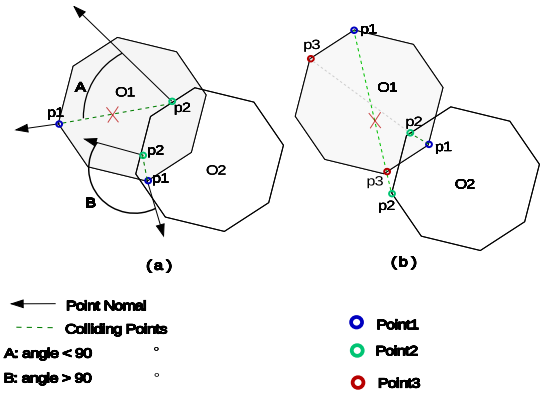


Figure 3.4: Colliding points validation. Point1, point2 and point3 denote the points identified by the Algorithm 3.

Figure 3.4(b) illustrates the second validation condition of a colliding point. We ensure that the point we found is inside O_1 using the same ray used to search a colliding point on O_2. But instead of tracing it inside O_2 it is traced against O_1. If the point found on O_1 (point3) is closer to the origin of the ray (point1) than the point on O_2, this collision pair is eliminated as the second point is outside O_1. The collision pairs that satisfy all the tests are kept to be treated by the collision response phase.

In theory this second test is not necessary, since we cannot hit an outward face from outside the object. However, in practice it happens often that objects are note completely watertight, and a ray can enter an object without touching an inward surface and hit directly an inward one. Also, due to floating point inaccuracy if a ray hits the border between two triangles it can happens that the outward surface is the first one to be touched, leading to a misdetection (this case is illustrated in Figure in 3.4(b))

3.3 Self-collision

To obtain realistic results when simulating highly deformable bodies, we need to avoid self-collision. Any primitive can potentially collide with any other primitive of the same object. In our case it means that we need to trace rays from every point in the mesh. Which tends to be more expensive than inter-object collision.

Self-collision can be detected using an extension of our method illustrated in Figure 3.5. For each point in the mesh we trace a ray following the direction of the point's normal. If this ray hits an outward surface of the same object we consider that there is a self-collision. A repulsion force is then set between these points, to separate these parts of the object.

As we trace rays from every point in the mesh, we need to quickly discard rays that will not cross any surface. By tracing outward rays we exploit the fact that an octree is usually much denser in regions near the object, than elsewhere. A ray that will not cross any other surface will quickly find a large cell that will take it out of the objects bounding box, and then be discarded.

Using our algorithm for self-collision detection offer a satisfying result in terms of quality. Unfortunately, using this approach for interactive simulation can be a limiting factor, since one ray is thrown from each vertex of an object which is computing costly. Selecting a subset of vertices, and not tracing

rays from all of them could be a solution to reduce this bottleneck, however the final result quality of this strategy is uncertain.

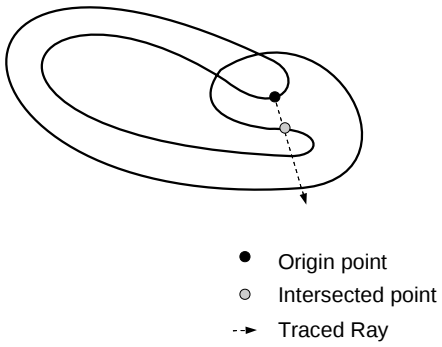


Figure 3.5: Self-collision detection: a ray is traced from the origin point in the direction of the normal; a self collision is found when it hits an outward surface (Intersected point)

3.4 Reaction

Once collisions are detected and modeled, we apply a penalty force to each pair of matching points. The force is proportional to the distance and the stiffness k of the involved objects. The direction is the same as the traced ray joining the matching points. It can be represented as:

$$f_{penalty} = k.\vec{P_1P_2}$$

The quality of the resulting force depends on the object smoothness. Sharp objects may undergo undesirable net tangential forces as illustrated in Figure 3.6. In these cases the force direction is not necessarily parallel to the normal of the collided object and some contact pairs can be considered more reliable than others. To reduce the effect of tangential forces due to the object sharpness we multiply the intensity of the force by the cosine of the angle α in Figure 3.6(b). This reduces the influence of the less reliable contact forces on sharp objects.

We tested the quality of the resulting forces created by our algorithm by observing the variation of the force direction using an object with different levels of smoothness. We employed a cylinder crossed by a plane as shown in 2D in Figure 3.7. The expected direction for the resulting force is a vertical force (Figure 3.7a). Due to symmetry, tangential forces should balance each other and the net tangential force should be null. However, due to surface discretization, the resulting force direction may differ from the normal of the plane as the rotational position of the cylinder changes.

To measure the variation of the resulting force, we tested cylinders with a number of sides ranging from 10 to 150. For each cylinder we took 100 different rotational positions, and measured the response forces. In Figure 3.8 we show the mean of the variations for a varying numbers of sides. As expected the quality of the resulting force increases with the object level of detail. For a cylinder having only 10 sides, we get a worst case deviation of 8%. As we increase the number of sides, variation levels decrease

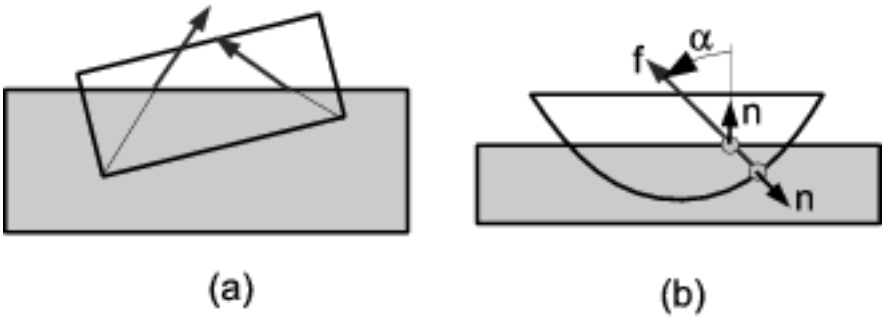


Figure 3.6: Contact force. In (a), a sharp object undergoes a non-null net tangential force. In (b), angle α is used to estimate the quality of the contact model and to weight its force.

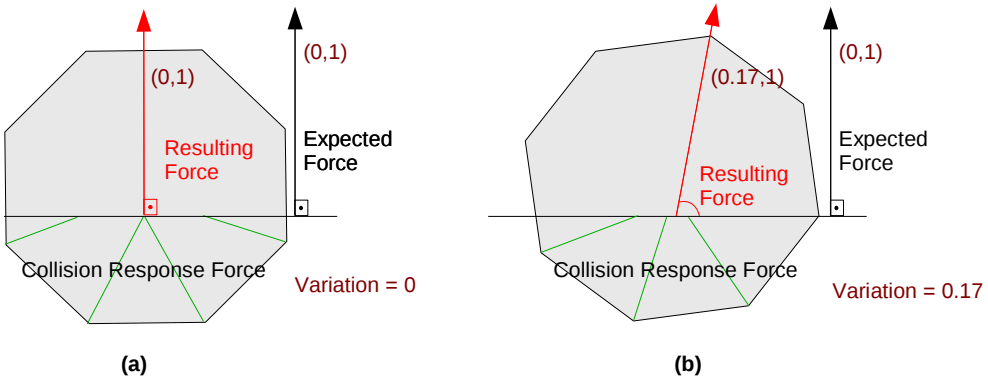


Figure 3.7: A cylinder undergoing various tangential forces due to low geometric resolution.

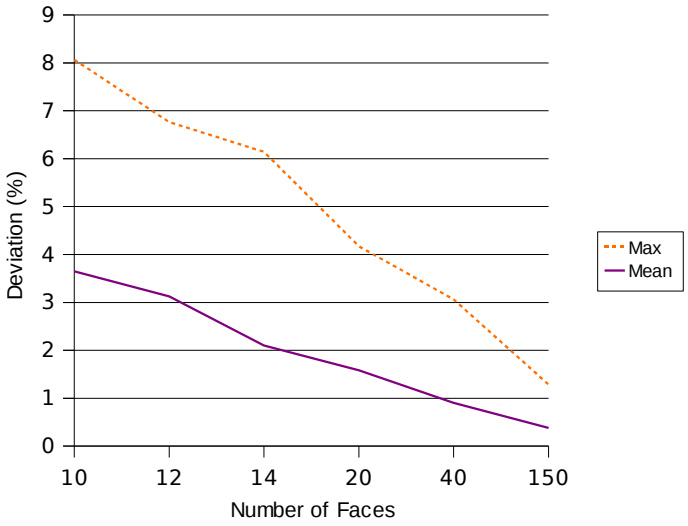


Figure 3.8: Ratio of tangential and normal force, against the number of cylinder faces.

to about 1%. This variation depends of course on the shape of the object, and the results improve with smoother objects.

An alternative to obtain a more reliable response is to use the object velocity to choose the force direction. A new collision response algorithm could be coupled just after our ray traced collision detection algorithm to consider this information. However this approach is hard to be exploited because penalty forces need sometimes multiple time steps to fully separate the objects. In these situations using the velocity direction has no meaning, because it was changed by the penalties forces in the previous step.

A limitation of our method occurs when a part of an object is completely inside the other in a non convex configuration. As illustrated in Figure 3.9, some of the traced rays will hit the outward surface (Point 2) of the colliding body (O_2) before hitting the collided one(O_1), In this case, no collision is detected and the contact force is null. Note that it does not induce instabilities, the only drawback is that the collision reaction will depend on the fewer rays that are not discarded, like ray A in Figure 3.9. The same happens when one object is completely inside the other, all the rays will be discarded and no collision force will be applied. In the same situation a proximity-based method would only succeed if the colliding vertices were not deeper than the proximity threshold.

Another limitation occurs when an object collides another object in a zone that is already in self collision. In this case we have an inward face inside the object. As a consequence the ray can be discarded if it hits this inward surface before hitting an outward surface.

3.5 Evaluation

We compared the efficiency of our method with a hierarchical implementation of a proximity-based approach, similar to the one proposed by [35]. The algorithms were implemented using the Simulation Open Framework Architecture (SOFA) [11]. This proximity algorithm uses a hierarchy of Axis Aligned Bounding Boxes to accelerate the collision detection. Also a parallel version of our algorithm

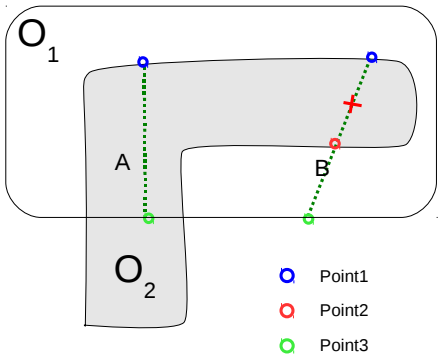


Figure 3.9: Our method can fail in case of non-convex intersection volume (Ray B).

was developed using KAAPI [55]. More details about this parallelization middleware can be found in Section 4.3.

The first test contains a scene where the initial state contains interpenetrating objects. From this initial state we observed how the algorithms manage to push the objects apart. Figure 3.10 shows the starting scene followed by the reaction produced by each algorithm.

The ray-tracing algorithm manages to detach the objects as it applies all the penalties in a direction that separates the objects. In contrast proximity-based algorithm tries just to push apart triangles that are too close. It creates some penalty forces that are oriented in a direction opposite to the one that should be used to separate the objects. As a result, only local deformations due to the penalty forces are observed, and the objects remain interpenetrated.

Even though this initial scenario cannot be reproduced in real life, it is useful in some special cases to be able to separate the objects that are initially interpenetrated. For example in surgery it is hard to define a scene where the organs boundary is well-defined with no initial intersection. In our case if the user creates such a scene we will be able to separate the interpenetrated objects, which is not the case for proximity algorithms.

Another advantage of our approach is the larger simulation step (dt) that can be used. With a large dt objects can move from a non colliding state to a deep interpenetration. With a proximity-based approach, deeply interpenetrated objects lead to triangles too far apart to be detected as colliding. Using the same scene with no initial intersection (Figure 3.11), the proximity-based algorithm gives satisfying results up to a maximum dt of 0.1 seconds, while our algorithm is still effective up to 0.45 seconds. Beyond this limits our algorithm fails because objects completely traverse each other, and only a continuous collision detection would be able to solve such case.

In the scene illustrated in Figure 3.12 we show a similar situation where the interpenetration of objects restrains the movements using a proximity-based approach. This behavior is due to large time steps that make the rings get interpenetrated. The proximity approach creates reactions forces that block the ring, preventing it to follow the chain movement. However, with the ray traced algorithm the rings can move freely, even with interpenetration. It allows to use larger time steps without degrading the

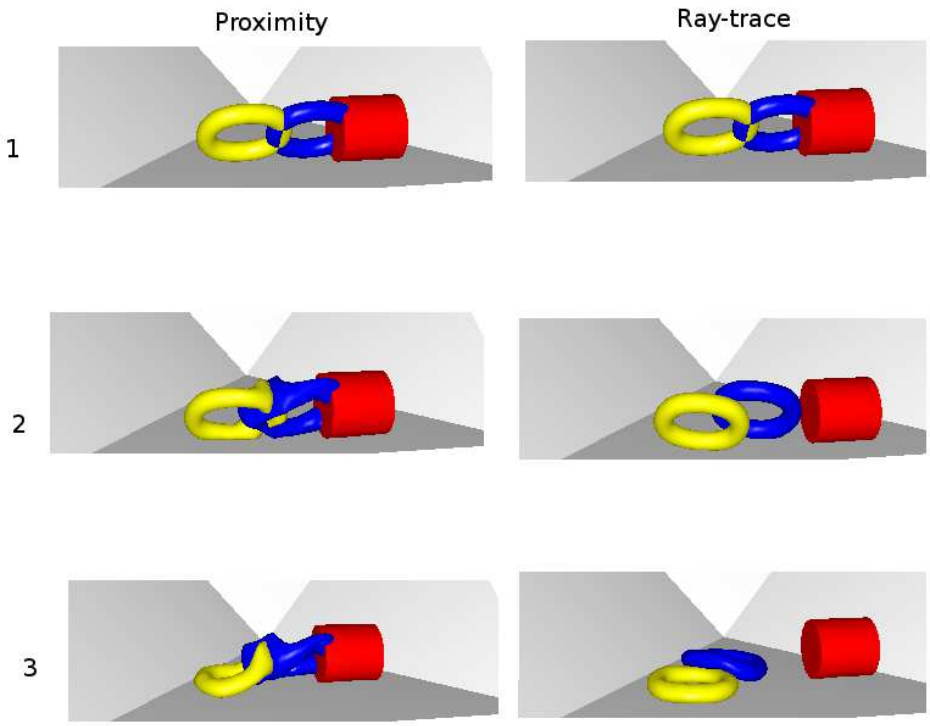


Figure 3.10: A test scene. Objects starts interpenetrated and we observe the evolution overtime.

expected behavior of the scene.

When simulating a scene like the one in Figure 3.11 on a Xeon 2.5Ghz machine the overall simulation using our algorithm outperforms by more than twice the proximity algorithm. Using ray traced collision detection we obtained 30 fps, while the proximity-based algorithm reaches only 12 fps. This performance gain is mainly due to a smaller number of colliding points detected by the ray tracing, as close triangles that are not in a colliding state do not generate colliding points. With less penalties applied, the solver runs faster.

In terms of scalability our algorithm behaves as expected, having a linear degradation of performance as the number of colliding objects increases. The basic element of our collision detection algorithm is the triangle. The algorithm performance depends directly on the number of triangles needing to be evaluated. In Figure 3.13, we display the time to solve 200 iterations with a varying number of objects in the scene. The objects used are tori initially interpenetrating each other.

3.6 Collision Detection Parallelization

To take advantage of the available multi-core architecture, we developed a parallel version of the collision pipeline. Pairs of colliding objects, and octree construction can be computed independently. We take advantage of this parallelism inherent to our algorithm to distribute the pairs to the different processing cores. At the beginning of the narrow phase we create one task for each colliding object that

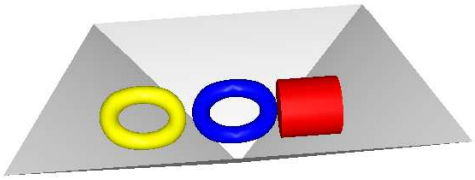


Figure 3.11: Scene used for performance comparison. Objects are not colliding at the beginning.

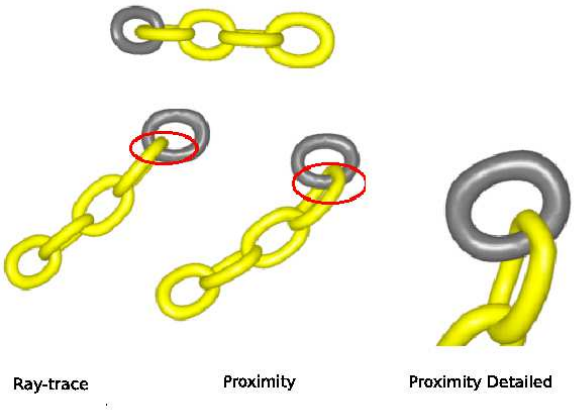


Figure 3.12: A deformable chain test.

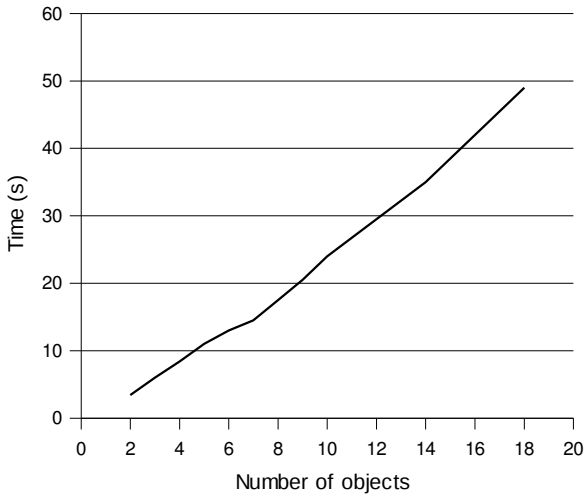


Figure 3.13: Performance evaluation with a variable number of colliding tori, each of them including 1600 triangles.

computes the octree. Then we create one task for each pair of colliding objects that executes our ray traced collision detection algorithm. However, a ray tracing task cannot be executed before the octree at both objects has been computed. This constraint is assured by a dependency link between these tasks, as shown in Figure 3.14.

To distribute the tasks over the processors we used a work-stealing algorithm, which works as follow. Each processor has a local list of tasks to be executed. When this list is empty the processor will try to steal tasks from the other. This kind of load balancing is necessary to avoid overloading a processor, which can happen when using a static work distribution. The reason for load unbalances comes from the non-uniform computational cost of the tasks, i.e. an intersection zone of a large number of primitives have a larger number of traced rays, and consequently takes more time to compute.

We tested this parallel implementation on a quad-core processor, using a scene containing 30 tori. The parallel version runs more than twice faster compared to a single core execution. The performance gain is limited by the remaining sequential computation as the penalty force creation were not parallelized.

Even if this parallel implementation was targeted at our ray traced algorithm, it can be applied to any other collision detection algorithm that contains a thread safe narrow phase test. This genericity is obtained by using coarse grain parallelization, as we only consider the parallelism among different pairs of colliding objects, without caring about the internal parallelization of a given algorithm. One example of fine grain parallelization of our algorithm is to trace each ray on a different processor. However such parallelization, even if potentially faster, is specific to a given algorithm, and cannot be reused by other collision detection algorithms.

This ray-traced collision detection algorithm was the first collision model integrated in SOFA that is robust to large time steps. Recently an image based collision detection algorithm has been integrated in SOFA [48]. It uses a rasterization technique that is fully implemented on Graphics Processing Unit (GPU), obtaining much faster results. We hope that the advances on ray tracing technique on GPU like presented by [7] would improve the efficiency of ray tracing on GPU, allowing to accelerate our

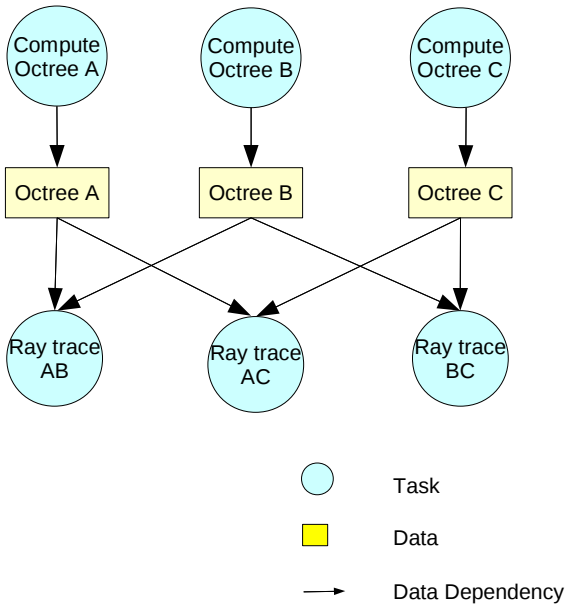


Figure 3.14: Example of data dependency graph for three colliding objects (A,B and C) when octree and ray traced collision detection are computed in parallel.

algorithm using such device.

3.7 Summary

In this chapter we described the first specific contributions of this thesis: a ray traced collision detection algorithm targeted at interactive simulation. We have shown that our novel collision detection and modeling approach is an interesting alternative to traditional proximity-based methods, especially for smooth deformable volumetric objects. The computation times are shorter, and the robustness allows us to apply larger time steps. The time spent by constructing an octree is compensated by the acceleration obtained on the ray tracing phase.

A parallel version of this algorithm, focused on coarse grain parallelism, was developed. The octrees of different objects are computed in parallel, and the tests on pairs of colliding objects were also parallelized. With this approach on a quad-core machine, the collision detection algorithm ran two times faster than a sequential execution.

This coarse approach allows to automatically parallelize any narrow phase algorithm containing a thread safe collision pair test. This approach involves a pure dynamic scheduling, that fits well to the collision detection step because most of the tasks are independent in terms of data access. However, a deeper knowledge of the execution structure is required to extract coarse grain parallelism from the rest of the simulation pipeline. This kind of problem will be addressed in the next chapter where we present a coarse grain parallelization algorithm that extracts parallelism from more complex execution structures, like time integration.

Part II

Parallel Physical Simulation

Chapter 4

Parallel Architectures and Programming

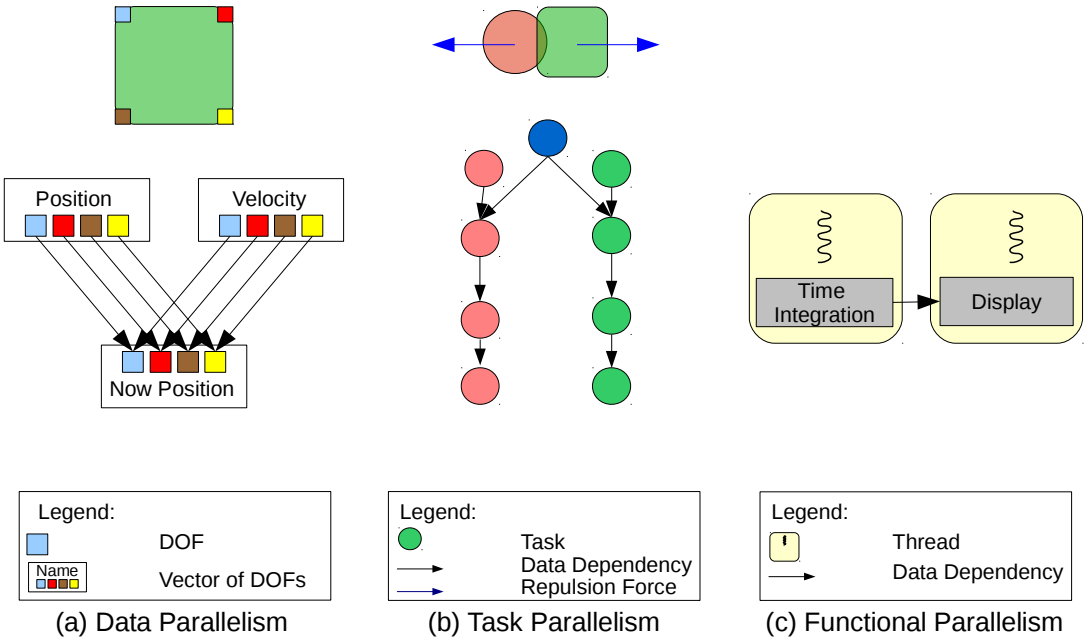
To take advantage of the new computer architectures and respect the interactive-time constraints when simulating more complex scenes, physical simulators have to increasingly rely on parallelism. However, writing parallel code increases the software complexity and imposes new strategical choices involving the target architecture and parallel tools. Examples of parallelism in physics simulation can be found at several levels. Fine grain data parallelism can be obtained when executing operations in parallel on all particles of an object (Figure 4.1(a)). A mid-coarse approach can exploit the parallelism between different bodies when updating their states (Figure 4.1(b)). In a coarse approach, different modules can execute in parallel, exploiting the functional parallelism between components such as visualization and time integration (Figure 4.1(c)).

We are here interested in a mid-coarse grain task parallelism, since at this level we can exploit the parallelism between different bodies without changing the internals of the physics algorithms implementation. By considering physics algorithms as black boxes and concentrating on the orchestration of task execution we can provide a parallelism that is transparent to the physics algorithm developer. It means that non experts in parallelism can write sequential code and at a same time still benefit from speedups on parallel machines.

To better understand the constraint and purpose of parallel architectures, we start by comparing and classifying them in Section 4.1. In Section 4.2 we present some works in parallel programming environments. In Section 4.3 we detail KAAPI, the parallel programming environment used during this thesis. Finally in Section 4.4 we discuss the way the existing interactive physical simulations use these parallel architectures and programming environments to respect the refresh rate constraint.

4.1 Parallel Architectures

In this section we review basic principles of parallel modeling and programming to provide the necessary background for non-experts in parallelism. We will start by presenting two complementary concepts to classify parallel architectures. The first one is the Flynn model, which groups parallel architectures accordingly to the number of simultaneous instructions and data streams seen by the running application. The second one differentiates parallel machines on the way the memory is distributed and accessed by



(a) Data Parallelism (b) Task Parallelism (c) Functional Parallelism

Figure 4.1: Three different levels of parallelization applied to a physical simulator.

the different processors. Finally we will map these classifications on existing parallel hardware used during the development of this thesis.

4.1.1 Architecture Models

Although there is no consensus in the parallel community, the Flynn taxonomy [50] is widely used to classify parallel architectures. This classification is based on the number of concurrent instructions and data streams available in the architecture. The four classes are: Multiple Instruction Single Data (MISD), Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). The former two classes are rarely considered in parallel computing: SISD describes a traditional *von Neumann* architecture where a single processor operates on data stored in a single memory. MISD is an unusual architecture, and most of the instances of such model are targeted at fault tolerant systems where multiple processors execute the same task redundantly over a single data stream.

Parallel computers classified as **SIMD** execute the same instruction over different data streams. The main advantage is their ability to manipulate large vectors and matrices in minimal time. For example if the size of a vector is equal to the number of available processors we can execute the operations on all the elements simultaneously. Usually this kind of architecture is restricted to fine grain data parallelism which reduces the range of problems that can be solved in parallel using this architecture. Examples of SIMD computers can be found in the early ages of computing. For instance, array or vectorial machines like ILLIAC IV and Cray-I. However the interest in SIMD waned in the late 90's when inexpensive scalar MIMD machines based on commodity computers became more powerful, allowing to construct more cost-effective and flexible supercomputers. Recently with the arrival of inexpensive programmable Graphics Processing Units (GPU), and data parallel programming environments like CUDA, the SIMD

architecture is again in the focus of the scientific computation community.

MIMD architectures employ multiple processors that operate on multiple independent data streams, making possible for developers to implement algorithms that require autonomous processors working asynchronously from one another. Eventual synchronization is done at software level, either by message passing or through data in shared memory. This asynchronous and decentralized architecture can be exploited by high level parallelism like task parallelism, or any algorithm where the instructions executed by a processor can diverge from the other. It is the most common architecture employed nowadays on high performance computing. Examples are multi-core and multi-processors, where each processing unit is independent and communication is done through shared memory. The largest supercomputers also use this kind of model based on clusters of multi-processors.

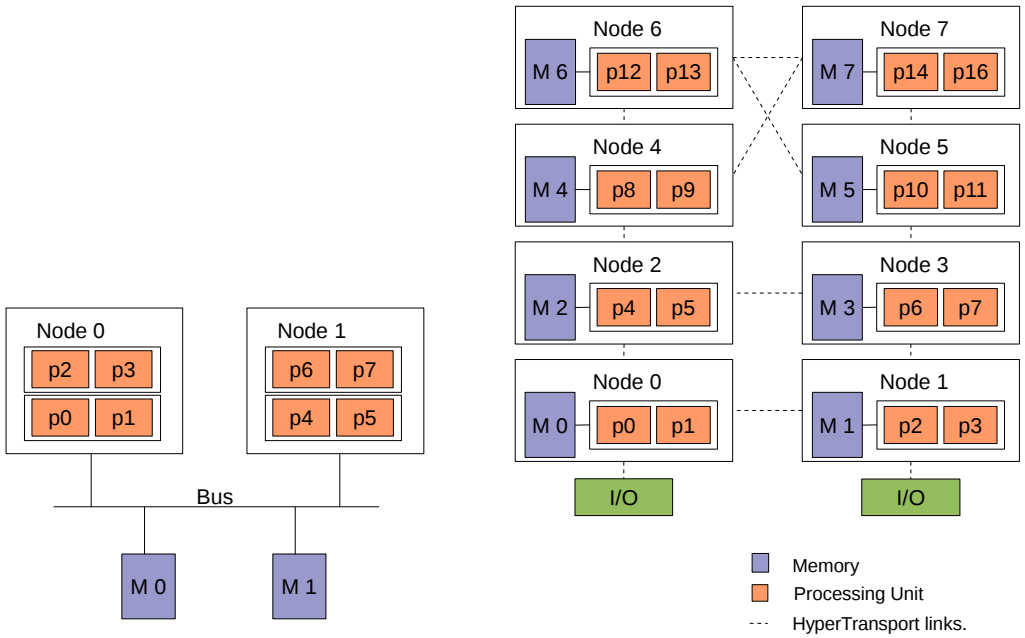
4.1.2 Parallel Computer Memory Architectures

Another way to classify parallel architectures is to consider memory organization and inter-process communication [73, 27]. The most prevalent nowadays are uniform memory access (UMA), non uniform memory access (NUMA) and no remote memory access (NORMA).

Multicomputer architectures, also known as **NORMA**, are computing units interconnected by a network. Each processor has its own memory and multiple address spaces exist. Unlike NUMA, each processor can only access its local memory unit. The communication with other processors must be done through the network.

In **UMA**, the cost of accessing any memory location is the same for all the processors. There is no differentiation between local and remote memory neither in the way it is accessed nor the latency cost. That is why the term Symmetric Multi-Processor is sometimes employed to refer to this group of machines. UMA architectures often are implemented using a centralized memory unit that is shared by all processors. In these case the central memory can easily become a bottleneck when multiple processors access the memory simultaneously. Uniform memory access was the most common approach in the first multi-core machines, since a centralized memory is easier to implement. It is for instance the case of the Intel Xeon *Clovertown* used for some of the performance benchmarks of this thesis . In this machine all the processors share the same communication channel (the Front Side Bus) to access the memory (Figure 4.2(a)), resulting on a weak scalability. Compared to NORMA, UMA architectures offers an easier way of programming since the address space is the same for all processors, i.e., a memory zone can be accessed from any processor, using the same memory pointer. However due to the bottleneck of a central memory, UMA architecture does not scale as well as NORMA.

In **Non-Uniform Memory Access** (NUMA), each processor, or group of processors has its own local memory. Like UMA, it offers a unified memory addressing space which facilitates the access to the data. Accessing data in a local memory is considerably faster than accessing data in a remote memory unit. Mechanisms to ensure the coherence across shared memory are used to simplify the way the programmer accesses the memory. Without such mechanism it would be impracticable to efficiently program these architectures, since it can contain several levels of memory: data can be located near the local processor, in the memory from another processor inside the same machine or, in the case of distributed shared memory (DSM), in the memory of a remote machine. Although it is harder to implement, it has some advantages when compared to UMA architectures. Providing separate memories for each processor reduces the performance hit that occurs on UMA architectures when several processors attempt to access



(a) Uniform Memory Access

(b) Non-Uniform Memory Access

Figure 4.2: Multi-processor architectures, with different memory access models

the memory at same time. This gain relies on the data locality and is proportional to the number of processors when all the data is found in the local memory. This architecture is more scalable than UMA enabling to associate a larger number of processors.

Examples of NUMA architecture are the AMD Opteron processors. In this machine the processors are interconnected using HyperTransport with the topology shown in Figure 4.2(b). A similar approach is also used by the Intel's Nehalem micro architecture, where the processors are interconnected by a network fabric similar to the HyperTransport, called QuickPath Interconnect. Both the Intel and AMD solutions implement a Cache Coherency protocol, that employs dedicated hardware to control the coherence of data locally cached by processors. This kind of functionality is a strong requirement for NUMA architecture, otherwise it would become prohibitively complex to explicitly control the coherence between data cached by different processors.

4.1.3 Specialized Coprocessors

Specialized coprocessors with high number of cores (*manycores*) became eminent due to high performance obtained with a relative lower cost when compared to general purpose processors. This lower cost is achieved using a large number of simple and specialized cores, offering a high level of data parallelism to the detriment of task parallelism. As a result the target applications for this architecture are linked to SIMD parallelism.

One example of specialized processor is the **Cell** architecture [61], which combines a modest performance general purpose processor with specialized units. The general purpose processor controls the

instruction flow, while the specialized processors ensure a high performance on data processing. The data communication among the specialized cores is explicit. Each specialized core entirely depends on Direct Memory Access (DMA) to transfer data to and from the main memory and other specialized cores. Cell processors are used from high performance supercomputers, like IBM Roadrunner [26], to Playstation 3 game consoles, which are also employed to scientific computation due to their lower cost [130, 84].

Graphics Processing Units (GPUs) are also a widespread class of specialized coprocessor. They are charged to accelerate the visualization and manipulation of graphics data. The acceleration is obtained using a massively parallel architecture to treat independent data, for instance computing a value for each pixel in the screen. Initially designed with a strict instruction set and limited functionality, these devices evolved to fully programmable architectures, extending their domain of applications to general purpose computing.

Historically the GPU architecture reflected in hardware the structure of standard graphics pipelines such as DirectX and OpenGL [33]. Up to 2006 two important groups of independent processing units [102, 94] could be found in the pipeline of 3D GPUs; the vertex processors which are in charge of the geometrical transformations, and the fragment processors, also called pixel shaders, that process pixels, performing operations such as applying texture.

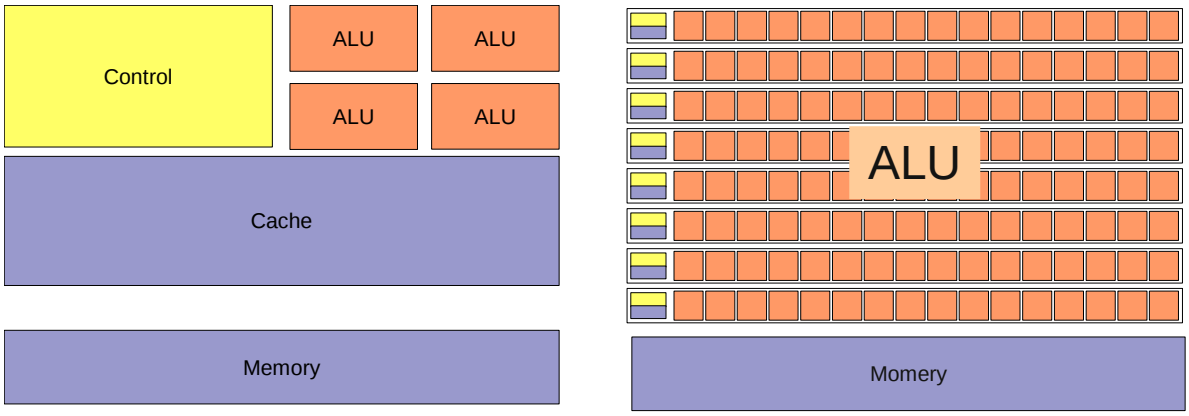
In the early 2000's these vertex and pixel processors became programmable, extending the possibilities for graphics developers. However, each group had different functionalities and implemented different operations, and consequently were not programmed in the same way. This architectural restriction, and the tools available for GPU programming restrained their usage almost exclusively to graphics applications.

The efforts and claims for general purpose computation, allied to the advances on microprocessor design resulted in a unified shader models. In this environment all the processing units have the same functionality, and are programmed in the same way. All the operations of the pipeline, are unified in an array of fully programmable stream processors.

In this direction, NVIDIA launched the G80 Series Graphic together with the Tesla devices [88] which are specially designed for high performance computing. A general purpose programming API called Compute Unified Device Architecture (CUDA) [100], is also available to this architecture.

These graphics cards dedicate most of the chip's area to arithmetical units instead of investing on cache and execution control as is done by general purpose processors. These arithmetical units are grouped in cores. The computations inside a core are performed in a SIMD mode. Multiple cores can be found in a single card operating independently from one another. The memory hierarchy is composed of three main levels. The shared memory, which is the fastest one, is shared by all the processing units of a core. Then the global memory that is shared by all the processing units in the graphic card, and can be used for communication between different cores in a card. Finally data can be transferred from the main computer memory, through a PCI-Express connection.

Based on general purpose architectures, the upcoming Intel's Larrabee [116] enforces the convergence tendency between specialized and general purpose processors. Larrabee offers a simplified version of traditional multi-core processors with common advantages such as cache management and branch prediction. At the same time each core can perform SIMD operations, as other GPU does. This architecture is compatible with the x86 instruction sets. Applications designed to run on multi-core architectures would execute in Larrabee without modifications. However, as in all GPGPU applications, not all soft-



(a) (b)

Figure 4.3: Architecture comparison between CPU and GPU. On CPU (left), most of the chip area is destined to cache and execution control, while on GPU (right) arithmetic and logic units are predominant.

ware will automatically benefits from utilization of vector processing units, and properly exploiting these units can require changes in the code.

Besides the potential speedup obtained using GPU, it adds more complexity to the application. The performance is strongly sensitive to the underlying hardware, and the lack of control and cache units can lead to poor data prefetching and branch prediction. Since this architecture is implemented as a set of SIMD processors, instruction divergences inside a processor (such as conditional branches) affects the performance, resulting in serialization of operations. Also by using an external device, we add another memory level to the system. Data must be transferred from the CPU memory to be accessed by the processing units, and this transfer time is not negligible. As a result, knowing the architecture internals is still a requirement for overcoming these restrictions and obtain good performance.

Even though it is harder to write code for specialized processors, some applications domains are intrinsically well fitted to data parallelism. Physical simulation is an example, since the virtual bodies are represented by particles or degrees of freedom, and the same set of operations is applied on all these data. This SIMD aspect of physical simulation usually results in a good exploitation of data parallel units. Many contributions have shown that, when exploiting this data parallel aspect of physical simulation algorithms, we can obtain impressive performance gains using GPUs [118, 39, 64, 63].

4.2 Parallel Programming Environments

In addition to the common constraint that a programmer must take into account when writing efficient sequential code, in a parallel environment the programmer must handle issues related to the larger number of resources available. Parallel programming libraries offer a high level interface for the underlying architecture, accelerating the parallel code development. This abstraction layer can also include other middleware functionality, such as work partitioning, load balancing, and data communication; which can differ from a parallel environment to the other depending on the target programming model and hardware. For instance we can classify the programming models in message passing, thread parallelism and data parallelism, accordingly to the way the data is managed.

A **message passing** application contains a set of independent communicating processes that are uniquely named and interact by receiving and sending messages to and from one another. Each process has its own address space, and accessing data from other processors can only be done by explicit message exchanges. Although diverging from the way the programmer accesses data in sequential programming, this explicit communication enforces the development of well structured parallel applications.

Thread parallelism, also known as shared memory task parallelism, is a model where different tasks are executed into a same process, sharing the same address space. Each thread has its internal data, and can access it independently from the other threads. However, memory zones shared among the threads can be used to communicate. The access to the shared data must be synchronized to avoid access conflicts. It can be done, for instance by using locks, barriers, or atomic operations. Some parallel environments can have a deeper evaluation of the data structure, coordinating the task execution in a lock free way and avoiding at the same time the access conflicts on shared data. From the programmers point of view, the main advantage of shared memory over message passing is to access shared data in the same way as the sequential programming language. Also it has a reduced software overhead, since accessing data can be done by directly using a pointer to the desired memory zone.

In **data parallelism**, the program is seen as a single instruction flow. Parallelism is obtained when a data parallel operations is performed on a set of data. For instance, operations in an array can be executed independently in a set of processors. In most of the cases the communication between processors, and the data transfer to each unit is implicit in the model. The programmers do not need to care about the data synchronization among the different data parallel processors. Data parallel applications are usually well adapted to SIMD architectures, since a same operation can be applied to multiple elements independently.

In the remainder of this section we will present examples of programming environments, with a special attention to shared memory task parallelism and heterogeneous parallel programming using data and task parallelism.

4.2.1 Message Passing

Parallel Programming Environments targeted on message passing, offer a set of communication patterns, that intend to ease the data exchange between processing units that are part of a same parallel machine. It is very popular on architectures like cluster or grids of computers.

4.2.1.1 PVM and MPI

The most popular specifications for message passing are PVM (Parallel Virtual Machine) [119] and MPI (Message Passing Interface) [60, 59]. Both system were designed to be portable, i.e., a program developed in an arbitrary architecture, can be recompiled and executed on a different architecture without changes in the code. Support to heterogeneous architectures is also present in PVM, and in some implementations of MPI (LAM, MPICH), it allows to machines with different architectures or operating systems to communicate, and collaborate in a same application. For this kind of heterogeneity the programming environment must provide communication directives that are charged to convert among the specific data representations.

4.2.1.2 Charm++

Charm++ is an object-oriented message driven parallel language that supports both shared and distributed memory. The parallel components in a Charm++ application are objects that communicate by message passing, following the concept of *actors*. Each actor contains sequential C++ code, and parallelism is obtained by executing operations on different objects in parallel. Differently from the traditional message passing concept composed by pairs of send/receive operations, in Charm++ when a message is received it is associated to an action, similar to the *active messages* concept [128].

Different load balancing strategies are supported. For instance graph partitioning is used for global load balancing, based on communication patterns. Also work stealing can be used to migrate objects between neighbor processors.

4.2.2 Shared Memory Parallel Programming

One advantage of shared memory architecture is the reduced overhead to access data. First because all the data are available locally, and there is no need to pay the cost of network transfer. It has also a reduced middleware overhead, since giving access to a data on shared memory can be as simple as a pointer to the memory. The spreading of commodity multi-core machines brings shared memory back to popularity. Together with the performance benefits, it is much easier for non-experts in parallelism to code using shared memory, since data access is similar to sequential coding. Simplifying the tasks of parallel programming is crucial for collaborative applications, where developers are not always experts in parallelism.

On the other hand message passing model leads to more structured and independent parallel tasks. The programmer is aware of all the communication needed to compute a parallel job, and will try to restructure the code to minimize this cost. Due to this trade off between easiness of programming and optimization, it is still a nowadays challenge to provide programming frameworks that allows to obtain optimal results on shared memory applications.

Plenty of parallel environments are targeted at multi-core/shared memory applications [37, 30, 110, 8, 21]. In the following sections we detail some of the most relevant ones, such as OpenMP, which is the most popular standard for shared memory programming. Then we will present Cilk, the programming language that promoted the usage of work stealing for task parallelism. We will also present Intel Threading Building Blocks, a promising parallel environment that is rising as an upcoming standard for task based parallelism. Finally we will give an overview of Athapascan, the parallel programming interface used during this thesis.

4.2.2.1 OpenMP

OpenMP is a programming interface for parallel programming on shared-memory multiprocessors. It defines a set of directives that extends C, C++ and Fortran programming languages. With OpenMP the programmer adds parallel directives to the sequential code. For instance we can mark a loop or a subroutine as a parallel zone and the work will be distributed among the threads. This kind of incremental parallelization cannot be achieved using multithreading interfaces such as POSIX threads, where the code

must be re-structured to define parallel subroutines. This non-intrusive aspect of OpenMP is the main reason of its nowadays wide acceptance.

Following OpenMP 3.0 [37], the programmer can define a parallel zone using three different concepts: parallel thread, parallel loops and tasks. In **parallel threads** the programmer defines a parallel zone, and all threads execute the same chunk of code in parallel. The threads can be differentiated using their identity, making possible to assign different rules for each thread, and implement a multi-task parallel execution.

The parallel thread execution follows a *fork and join* model where a master thread sequentially executes the main code, while a set of *worker* threads are used to collaborate with the master thread on the parallel parts of the code. The scheduler will assign these tasks to the worker threads. At the end of the parallel section the main thread waits for all the worker threads to finish before continuing the sequential code.

Loops iterating over independent data can be parallelized using **work sharing** constructs. Using work sharing, a range of the computing data is assigned to each thread. Instead of executing the iterations sequentially, each thread will operate in a subset of indexes in a data parallel way. To avoid conflicts on parallel execution, the user can tell the compiler which data are private to the thread or shared among threads. The user can also choose the algorithm employed to assign data to the threads. For instance, a static scheduling will allocate equal data chunks to all the threads at initialization time. Using dynamic scheduling, only a part of the work is assigned to the threads, the remaining data is then computed by the threads as they become free. It results on a better load balance, specially when the work load is not uniform.

Recently, a parallel task directive was introduced [20]. It spawns a task that will be executed asynchronously by an available thread. Unlike parallel thread constructs where all the threads executed the same work, the code zone identified as a task is executed asynchronously by only one thread. A task can recursively spawn other tasks, extending the application domain of OpenMP to recursive parallelism, such as branch and bound algorithms. The scheduling algorithm is not standardized, allowing the implementation to choose the employed algorithm [46]. One example of scheduling algorithm that is well adapted to task parallelism is Cilk's work stealing, which will be presented in the next Section.

4.2.2.2 Cilk

Cilk is a multithread runtime system that extends the C language with special keywords to create and synchronize parallel tasks. The programmer is in charge of exposing the parallelism by identifying elements that can be safely executed in parallel. On the other side, the run-time environment is responsible of implementing the scheduling policies. The keyword `spawn` indicates that a procedure can safely operate in parallel with other executing procedures. The keyword `sync` indicates that the current execution cannot proceed until all previously spawned procedures have completed.

A work stealing approach is used for load balancing. When a processor finishes its work it will try to steal some work from the other processors. Cilk work stealing algorithm has a provable performance in a sub-class of fork and join model, called *fully strict* (well structured) model. Blumofe et al. [30] defined the notion of fully-strict computation as follow. "A **fully strict** program is one in which a parent task creates a set of child tasks that are completely parallel to one another. The only allowed arguments trans-

fer and synchronization are between the child task and its parent". The execution of a Cilk program can be represented as a Directed Acyclic Graph, composed by the spawned tasks linked to its corresponding predecessors. This fully strict parallelism is well adapted to express recursive algorithms.

To show an example of a Cilk code, we take a recursive implementation of a Fibonacci number computation. The sequential reference algorithm is in Figure 4.4(a), it is not mean to be an optimal way to compute the Fibonacci number due to duplicated computation, but it is a good way to show how we can take profit of parallel machines with few changes in the code. Additionally, to better compare Cilk with other programming environments (cf. 4.2.2.4), we introduced two new operations that uses the intermediary results of each side of the recursive call (Figure 4.4(b) lines 10 and 11).

To enrich the original function using Cilk, the `fibonacci` function is marked as a task (cf. `cilk` keywords in Figure 4.5). This task is spawned for each recursive call to `fibonacci`, to compute $n-1$ and $n-2$ inside the function body. A sync operation is added just after the tasks that compute $n-1$ and $n-2$ to wait for them and sum their result.

```
1  int fibonacci (int n)
2  {
3       if (n < 2)
4          return n;
5       return fibonacci (n-1)
6              + fibonacci (n-2);
7  }
```

(a) Recursive Fibonacci number computation.

```
1  int fibonacci (int n)
2  {
3       int x,y;
4       if (n < 2)
5          return n;
6
7       x = fibonacci (n - 1)
8       y = fibonacci (n - 2)
9
10      uses (x);
11      uses (y);
12
13      return  x + y;
14 }
```

(b) Modified implementation.

Figure 4.4: Sequential Fibonacci algorithm used to compare different parallelization approaches.

Each `fibonacci` task has a `uses` task associated to it. Since in a fully strict paradigm, a task is only allowed to synchronize with its parent, we cannot specify that a `uses` task only needs to wait for its respective `fibonacci` task. For instance, this restriction prevents a parallel execution between lines 11 and 16 in Figure 4.5 even if there is no data dependency between these tasks. This kind of situation can be better exploited using data dependency analysis, as will be explained on Section 4.2.2.4.

4.2.2.3 Threading Building Blocks

Threading Building Blocks (Intel TBB) [110] is a library developed by Intel Corporation for writing software programs that take advantage of multi-core processors. It is developed using C++ templates and is a purely standard C++ library. It means that unlike OpenMP and Cilk, it does not require special languages or compilers. Threading Building Blocks is focused on data parallel programming. Most of the data parallel strategies rely on generic programming using templates by reimplementing some standard algorithms. But in addition, it fully supports nested parallelism, like Cilk. A work-stealing load balancing scheduling is used to redistribute the work across the processors.

Using TBB we can practically parallelize the same class of algorithms that are efficient on Cilk.

```
1  cilk void uses (int n){
2      /* ... */
3  }
4   cilk int fibonacci (int n)
5   {
6       if (n < 2) return n;
7       else
8       {
9          int x, y;
10
11         x = spawn fibonacci (n-1);
12         y = spawn fibonacci (n-2);
13
14         sync;
15         spawn  uses(x);
16         spawn  uses(y);
17
18         return (x+y);
19      }
20  }
```

Figure 4.5: Fibonacci number computation using Cilk. Each call to spawn creates an asynchronous tasks that can be executed in parallel. The sync keyword is used to wait all the previously launched task and sum their result.

Historically Cilk is a library that strongly inspired the development of Intel TBB and most of its execution environment concepts are inherited from Cilk. Other libraries, like STL [97], STAPL [13, 107, 14] and OpenMP, developed concepts that are part of the TBB bases.

The Intel TBB implementation of Fibonacci number (Figure 4.6) is similar to Cilk's one. Task are spawned but synchronization are required to wait the results of the child tasks. The main difference is in the programming interface, since Intel TBB is a C++ library it requires much more code to express the same functionality than Cilk, which is a compiled language.

4.2.2.4 Athapascan/KAAPI

Athapascan [54, 87] is a parallel programming interface developed by the MOAIS project from the INRIA/LIG. It relies on data dependencies expressed using explicitly shared data types (by default all data are local). Like Cilk, it supports recursive parallelism to express divide and conquer algorithms. Additionally, Athapascan can exploit non-recursive parallelism by means of data dependency analysis.

A runtime environment, called KAAPI, offers an abstraction of the hardware architecture, providing a uniform interface for parallel programming and communication. It also offers fault tolerance and load balancing support [1].

The load balancing uses a work stealing approach like done by Cilk, where an idle processor steals work from busy processors. The high level language used to express data dependencies is inspired by the Jade Parallel Programming Language [85, 113]. Two keywords are employed: `shared` and `fork`: The `shared` keyword declares an object in the global memory, which can be accessed by any processor. The `fork` keyword creates a new parallel task, similarly to the `spawn` construct of Cilk.

[1] KAAPI will be detailed in Section 4.3, Page 52

```
class Uses: public task {
public:
    int  value;

    Fibonacci( int value_ ) : vamue(value_) {}

    task* execute() {
        /* ... */
        return 0;
    }
};

class Fibonacci: public task {
public:
    int n;
    int*  sum;

    Fibonacci( int n_, int* sum_ ) : n(n_), sum(sum_) {}

    task* execute() {
        if ( n < 2 ) {
            *sum = n;
        }
        else {
            int x, y;
            Fibonacci& a = *new( allocate_additional_child_of() ) Fibonacci(n-1,&x);
            Fibonacci& b = *new( allocate_additional_child_of() ) Fibonacci(n-2,&y);

            spawn( b );

            spawn_and_wait_for_all( a );

            *sum = x+y;

            Fibonacci& c = *new( allocate_additional_child_of() ) Uses(y);
            Fibonacci& d = *new( allocate_additional_child_of() ) Uses(x);

            spawn( c );

            spawn_and_wait_for_all( d );
        }
        return 0;
    }
};
```

Figure 4.6: Example of Fibonacci number using Intel TBB

A task in Athapascan is a function whose signature contains all the data it shares with other tasks. This signature also contains the access mode of each parameter : a read-only parameter is declared using `shared_r` while a write-only data is declared as `shared_w` The program execution is driven by the data availability, accordingly to these access modes. A task requesting a read access must wait for all the previous writers before starting the execution.

Combining recursive and data driven execution one can express more complex structures and better exploit the parallelism of the algorithms. One example using the Fibonacci number computation is shown in Figure 4.7. As done in Cilk implementation, the Fibonacci number is computed recursively, and tasks are spawned to compute each branch. However, the execution is controlled by the data flow graph and not by the program recursion structure of the full strict model (Section 4.2.2.2).

The unrolling of the data flow graph is done as follows. The initial task is called by the main thread

```
struct sum
{
    void operator () (a1::Shared_w < int >  result,
                      a1::Shared_r < int >  subresult1,
                      a1::Shared_r < int >  subresult2)
    {
        result.write (subresult1.read () + subresult2.read ());
    }
};

struct uses
{
    void operator () (a1::Shared_r < int >  value)
    {
        uses(value.read());
    }
};

struct fibonacci
{
    void operator () ( a1::Shared_w < int >result, int n)
    {
        if (n < 2)
        {
            result.write (n);
        }
        else
        {
            a1::Shared < int >  res1;
            a1::Shared < int >  res2;

            a1::Fork < fibonacci > () (res1, n - 1);
            a1::Fork < fibonacci > () (res2, n - 2 );
            a1::Fork < uses > () (res1);
            a1::Fork < uses > () (res2);
            a1::Fork < sum > () (result, res1, res2);
        }
    }
};
```

Figure 4.7: Example of Fibonacci number using Athapascan

and receives the output buffer as parameter(Figure 4.8(a)). When the initial tasks is executed it creates other tasks recursively (Figures 4.8(b) 4.8(c)). Inside the `fibonacci` function the tasks are linked by data dependencies. For instance the `sum` tasks waits the end of both preceding `fibonacci` functions, while the `uses` tasks only wait for the end of their respective `fibonacci` tasks.

Using data-flow graph offers a more flexible parallel programming compared to Cilk and Intel TBB fully strict model. Using Athapascan we can have a synchronization between tasks that are at a same recursion level. For instance, tasks `sum` and `fibonacci` have a data dependency to compute the sum of both `fibonacci` tasks. To support synchronization between tasks at a same level, TBB and Cilk provides a synchronization task, which forces the application to wait for the end of all the previous spawned tasks. This synchronization is used to wait for all the previous `fibonacci` tasks before summing them. In fibonacci computation, this synchronization call does not impairs performance, since in all the cases the summing depends on all the previous `fibonacci` tasks. However we can see the limits of such approach when adding a task that does not depends on all the previous ones, for instance a `uses` task that uses the intermediary result of each `fibonacci` task.

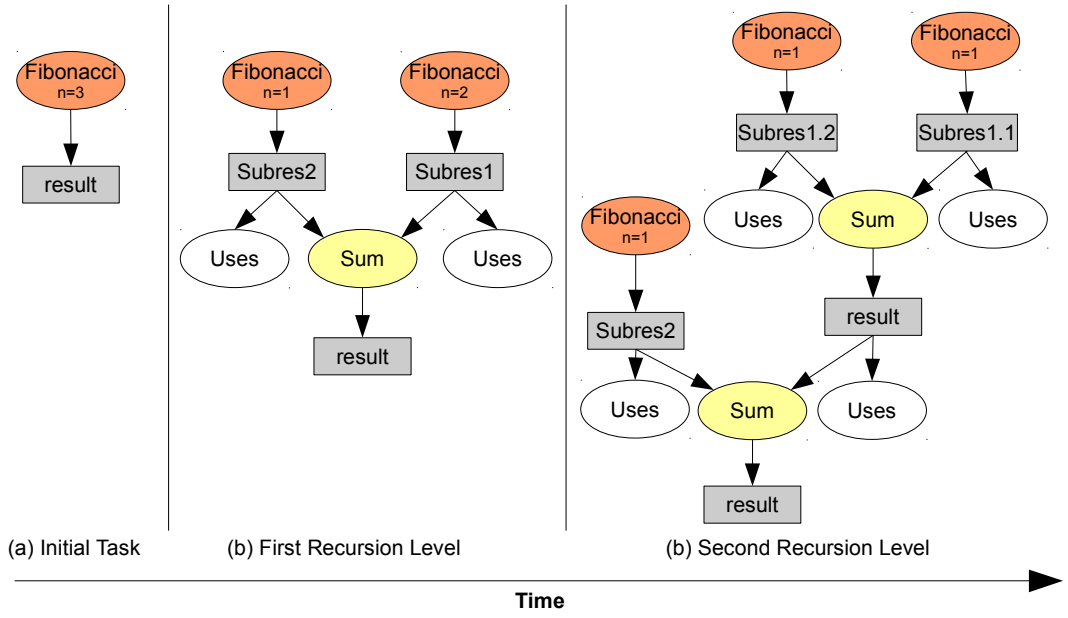


Figure 4.8: Fibonacci execution graph

Using a data dependency analysis we can execute the `uses` task once the respective `fibonacci` task has ended. On the other hand, using a purely recursive approach, a synchronization is needed to wait for all the `fibonacci` task, as happens in Cilk Fibonacci example (Figure 4.5, Line 14).

Load balancing using work stealing for non-recursive algorithm is efficient if the amount of stolen work is considerably high. Otherwise, stealing light tasks will not keep the thief processor busy for a long time and result in frequent stealing trials. Too much stealing activity increases the overhead of the runtime environment and impairs performance.

To efficiently execute non-recursive parallelism and increase the amount of independent work assigned to each processor, KAAPI also supports static scheduling. The data flow graph generated by the Athapascan interface is partitioned and mapped to processors using external libraries such as DSC [132], METIS [80] and SCOTCH [103]. This kind of approach can exploit non structured parallelism, where parallel zones are not explicitly isolated in the code. Another important class of algorithms that benefits of this approach are iterative algorithms, where the same set of tasks is executed repetitively. For these applications the graph partitioning is computed only once at the beginning, and redeployed on each iteration step, which helps amortizing the static scheduling cost.

By supporting all these execution models Athapascan can efficiently be used for a wide range of applications. The usage of one approach does not prevent the other, allowing to combine static and dynamic scheduling in a same application.

4.2.3 Programming Environments for Heterogeneous Architectures

The development of parallel algorithms for general purpose architectures is provided with a variety of well established API like these presented in Section 4.2. However, programming for specialized processors is usually much harder. One example is the usage of graphics cards for general purpose programming. For a long time the only way to exploit graphics cards was to artificially map the algorithm to the graphics pipeline, by considering, for instance, variables as pixels or vertices and reading the result from the computed texture image.

4.2.3.1 Architecture-specify Code Generation

Recently, new programming interfaces have been introduced to assist the development of general purpose programming for graphics cards. On example is the CUDA API, which offers a programming language based on C with special keywords to specify specialized functions. Each CUDA function, called **kernel**, represents the data parallel operations to be executed in a GPU. The same kernel is automatically mapped to several processing units and is executed in a SIMD way. Despite the fact that the load distribution across the processing units is transparent, the user is in charge of explicitly controlling the data transfer between GPU and CPU and the different memory levels of the GPU. CUDA offers a programming language that is much similar to what general purpose programmers are used to, compared to hardware based programming. However, developing for specialized processors is still hardly maintainable. AMD offers a similar development kit for their ATI-based GPUs called Stream SDK. Cell processors have their own programming interface. These interfaces work exclusively on their vendors GPUs with no compatibility among them.

A joint effort from the computing industry resulted in a new standard interface called OpenCL, a programming interface that allows reusing code on different platforms such as CPUs, GPUs, and other processors. It offers a programming structure that is similar to CUDA in terms of language and memory hierarchy. The OpenCL open specifications enables the support of a larger number of architecture when compared to proprietary interfaces such as CUDA. On the other side an OpenCL code is hardly efficient on multiple architectures. The way the code is usually written tends to favor GPUs to the detriment of multi-core machines.

HMPP [45] also offers multi-architecture code generation. A set of OpenMP-like directives is used to specify the way data is accessed by the specialized device and which are the target architectures that are well suited to run a given chunk of code. However it only offloads work to a specific processor and no deeper analysis of data dependency, or load balancing is done.

4.2.3.2 Scheduling on Heterogeneous Architectures

Another complementary group of tools focus on the scheduling on heterogeneous architectures, leaving to the user the task of generating architecture-specific code. Support to GPU has being added to Charm++ [79] [129]. In this implementation the user defines if a task will be executed by the CPU or by the GPUs. Using the Charm++ interface the tasks are offloaded to the graphics unit.

In StarPU [18, 17], the specialized code provided by the programmer is wrapped in task. It employs

a dependency graph to determine when a task is ready to execute. The ready to executed tasks are then scheduled to execution. In addition, a data management library is charged to minimizing the transfer among devices. A filtering interface allows to select only the needed sub-part of data, avoiding to waste time transferring the whole data. They experiment various scheduling algorithms, some enabling to get "cooperative speedups" where the GPU gets support from the CPU to get a resulting speedup higher to the sum of the individual speedups. Load balance criteria involves cost models and performance prediction in order to better chose the target processing unit that minimizes termination time. Published experiments include tests with one GPU only.

We know two different approaches for multi GPU dynamics load balancing. The extension of the StarSs for GPUs [19] proposes a master/helper/worker scheme, where the master inserts tasks in a task dependency graph, helpers grab a ready task when their associated GPU becomes idle, while workers are in charge of memory transfers, launching the execution and retrieving the results. The master leads to a centralized list scheduling that work stealing enables to avoid. RenderAnts is a Reyes renderer using work stealing on multiple GPUs [135]. The authors underline the difficulty in applying work stealing for all tasks due to the overhead of data transfers. They get good performance by duplicating some computations to avoid transfers and they keep work stealing only on one part of the Reyes pipeline. Stealing follows a recursive data splitting scheme leading to tasks of adaptive granularity. Both RenderAnts and StarSs address multi GPU scheduling, but none include multiple CPUs. During our work we intend to study the necessary mechanism to combine multiple CPUs and multiple GPUs in a single simulation.

4.3 Previous Works on Athapascan/KAAPI

The main difference between Athapascan and parallel environments like Cilk [106], Intel TBB [109] and OpenMP [40], is the Athapascan Data Flow Graph (DFG) representing the dependency between tasks and data. By evaluating this DFG we can avoid spurious synchronization barriers present for instance in the SOFA recursive attempt (Section 5.1.1) or in the Fibonacci example (Section 4.2.2.2).

4.3.1 KAAPI Structure

KAAPI is a runtime environment that implements the Athapascan API. It offers an abstraction layer for the underlying hardware. KAAPI provides data sharing, fault tolerance and task scheduling support among other assets. Among all these functionalities of KAAPI the discussion in this section will be restrained to the scheduling aspects that are used during this thesis.

In Figure 4.9 we show the hierarchical structure of KAAPI. It is organized as follow. One KAAPI process is launched per computing node. Then, each KAAPI process is composed by system threads called K-Processors. Usually there is one K-Processor per processor in the machine. The K-Processors are in charge of executing the user level threads implemented by KAAPI. These lightweight threads are called K-Threads, and contain a stack of tasks spawned by the user. The K-threads are non preemptive, i.e. their execution cannot be interrupted by the runtime environment. This kind of hierarchical structure is used because creating and switching context among KAAPI threads is much faster than switching context on system threads.

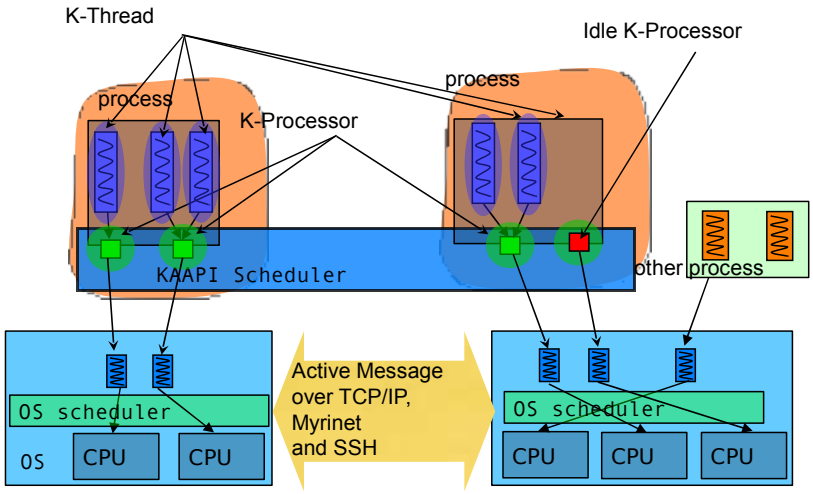


Figure 4.9: Hierarchical structure of KAAPI

4.3.2 Lightweight Thread Execution

To be unscheduled a K-Thread must explicitly yield the processor. It happens in two situations: when there is no more task to execute or when it cannot go further because it is waiting for a condition to be satisfied, due to data dependencies for example. If there is no more tasks to execute, once unscheduled the thread is freed. On the other hand if the thread is in a waiting state it will be inserted in a list of waiting tasks. This thread will eventually be rescheduled once it is in a ready state.

Usually the state of a K-Thread depends on the state of the first task in the stack. The possible states for a task are:

- **Ready:** The initial state of a task when created by Athapascan. All task structures are ready to execute. If a thread finds a task in a ready state, and all the data needed are produced, it can execute it.
- **Waiting:** Tell the runtime environment that the task is waiting for an external signal, and cannot be executed, even if all the dependencies are satisfied.
- **Executing**: The task is currently being executed. This state can be used, for example, by a thief thread to find tasks to steal and skip tasks that are being executed.
- **Finished:** Flags a task as already executed, avoiding, for instance, to execute the same task multiple times.
- **Stolen:** Signals that the task has been stolen by another thread. This task cannot be considered as finished until the thief thread did not signaled the task termination.

KAAPI considers the task creation order as a valid execution order. If the first task in a K-Thread is in a proper state (Ready), it executes the task without looking at the data dependencies. This assumption avoids the overhead of verifying the state of each data before executing a task.

The data-flow graph given by Athapascan is used to ensure the conformity with the sequential execution in the case of stealing. When a stolen task is found, the execution of the remaining tasks is conditioned by the data dependency analysis. While the stolen task is not flagged as finished we can only execute the tasks that do not depend on data produced by this stolen task.

4.3.3 Dynamic Load Balancing

At the beginning of the execution only the main K-Thread has tasks on its local stack and all the other K-Threads are waiting for jobs. To redistribute the tasks among the K-Processors, KAAPI implements a work stealing approach [32] following the example of Cilk. A working K-Processor becomes a thief when it has no more K-Threads to execute or when all of them are not ready to execute.

An idle K-Processor randomly chooses another K-Processor as a victim, and tries to steal work from it. Stealing is done in two levels. First the thief searches for K-Threads ready to execute and steals the whole K-Thread. A K-Thread is considered ready if its first task is in a ready state. The second level is needed when there is no ready K-Thread. In this case, the victim inspects the tasks inside a K-Thread of the victim processor to search for a ready task. In the first case the idle processor steals a whole stack of tasks, while in the second case it steals only a task.

With a two levels stealing an idle processor can steal the largest amount of work when possible (K-Threads), and only stealing a smaller grain (tasks) when strictly necessary. However, in parallel recursive applications one task spawns other tasks. Stealing a task in this context also means stealing all their children tasks. Taking a task high in the hierarchy usually results in stealing more work than stealing from a terminal task. This variable grain also helps reducing the number of stealing trials. At the beginning of the execution large amounts of work would be stolen, and the K-Processor stays busy for a long time before a new stealing trial, while at the end of the execution the majority of the task are terminals and processors collaborate to finish this remaining work.

4.3.4 Static Load Balancing

To be efficient the number of thefts during the execution must be low. Even with a proven efficiency in the case of fine grain fully strict algorithms [32], the work stealing approach results in a high execution overhead for iterative algorithms.

Iterative applications have a *flat* data flow graph, i.e. all the tasks are terminal and do not spawn any other task. Consequently individual task stealing will result in a large number of steals since a small amount of work is obtained at a time. To offer a better scheduling for iterative applications, Laurent Pigeon [104] extended KAAPI with a static scheduling approach. The application domain studied is the Computer Aided Process Engineering (CAPE) [22]. The proposed solution adds a preprocessing step to the execution. This preprocessing can be subdivided in five phases (Figure 4.10): graph extraction, scheduling, partitioning, thread generation and deployment. The goal of this preprocessing is to generate as many partitions as available processors, to evenly redistribute the work. Those partitions represent groups of tasks that execute independently, with no central coordination.

The **graph extraction** (Figure 4.10(b)) is performed through the Athapascan API. The difference is in the way the program is executed. Instead of running the tasks as they are created, KAAPI starts by

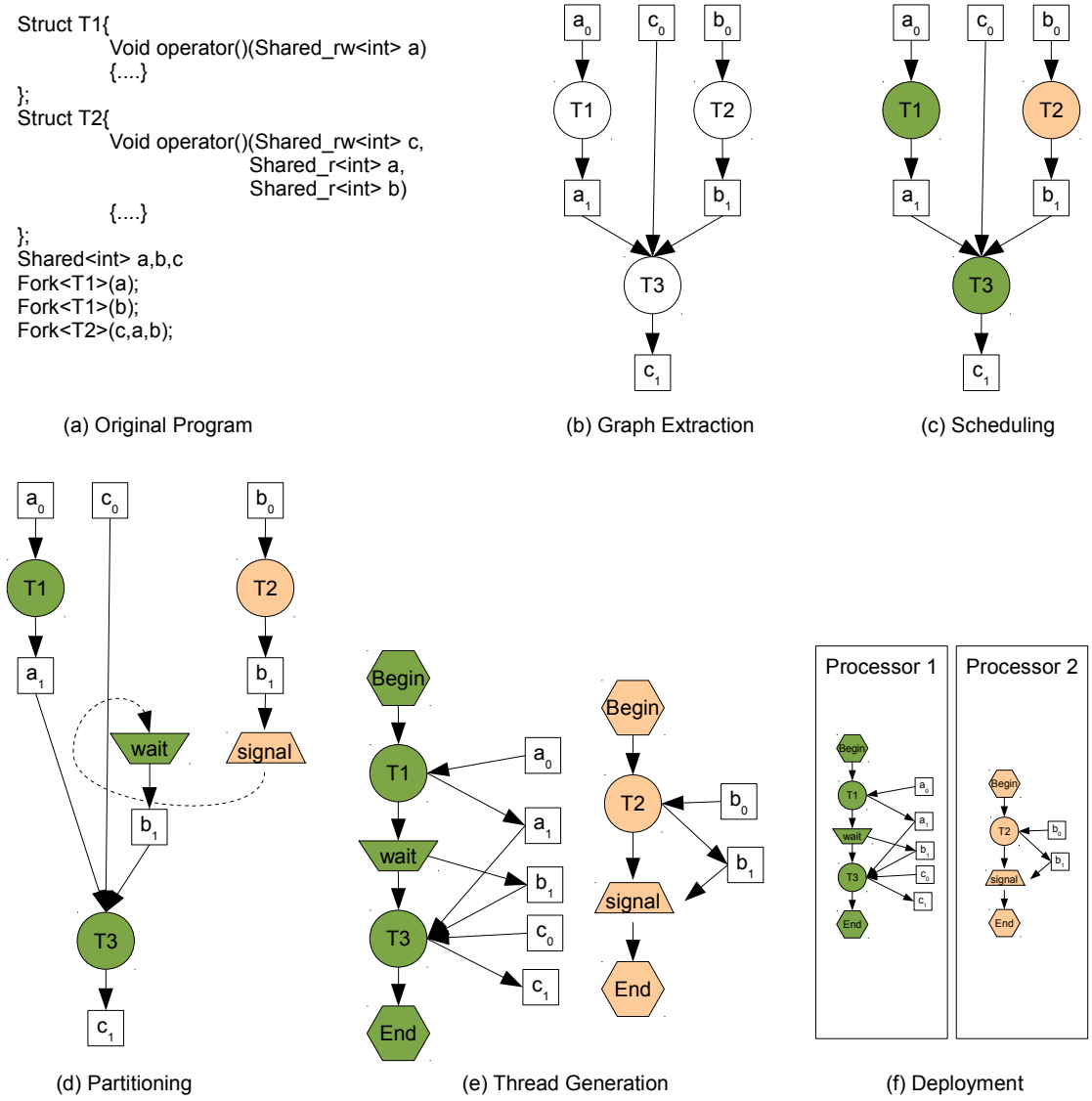


Figure 4.10: Steps of static partitioning in KAAPI: From the original program (a) a data flow graph is extracted (b). Each task is mapped to a partition (c). The task are partitioned and the graph enriched with communication tasks (d). Individual stacks of tasks are generated (e). The stacks of tasks (K-Threads) are distributed across the processors (f)

only creating a task graph, without executing them. At the end of this first step we have a dataflow graph that represents the entire application execution. This graph will then be used as input to the scheduling step.

The **scheduling** (Figure 4.10(c)) step colors the task graph by assigning an execution site to each task in the graph. The set of tasks assigned to the same execution site is the equivalent to a partition. This step relies on an external scheduler that computes the best distribution accordingly to a given criteria. The supported schedulers are Dominant Sequence Clustering [132], the METIS package [80] and the Earliest Task First method [72]. Usually the input of the scheduling step is the dependency graph and the number of desired partition. The output is the initial graph with the tasks assigned to the partitions.

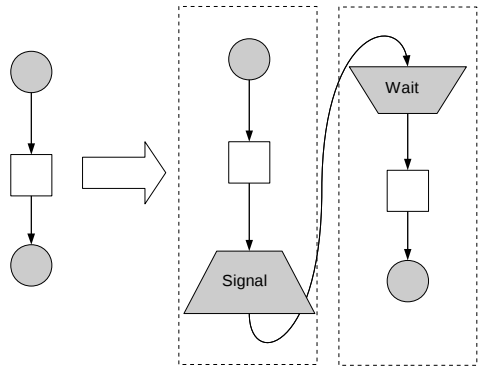


Figure 4.11: Control tasks employed to guarantee data access coherence between different partitions. The reader waits for the data to be produced by the writer.

During the **partitioning** (Figure 4.10(d)) step, we enrich the DFG by adding special tasks that are charged to synchronize different partitions. If a task needs to access some data produced by a task in different partition we add control structures to signal when the data are ready. In a Writer/Reader profile, as shown in Figure 4.11, a *signal* task is inserted after the writer task. This task signals all readers when the data are ready. A *wait* task is placed just before the first reader of each partition, to wait passively for the signal. These additional tasks are needed because KAAPI considers that the task order in a K-Thread is a valid execution order. However this assumption is not valid after partitioning, as the data producer and the data reader will be in a different K-Thread. That is why the partitioner needs to add these task to force the synchronization between threads.

The partitioning creates a single DAG containing the additional data synchronization tasks. From this graph the **thread generation** (Figure 4.10(e)) step creates one different subgraph for each partition. It means that after this step we will no more have a single DAG representing the whole application, but a set of autonomous DAGs linked by the data synchronization tasks (*signal* and *wait*).

The **deployment** (Figure 4.10(f)) distributes the partitions over the processor. As the number of partitions depends on the strategy chosen by the user, there is no guarantee that there are as many partitions as processors. Usually the scheduler will return more partitions than the available processors. In this situation, the partitions are distributed using a Round-robin algorithm.

The results presented during Laurent Pigeon's thesis show that the overhead added by the static scheduling is small when compared to the overall execution time. For instance, to schedule about 60,000 tasks it takes less than a half second, while the timestep execution takes more than ten seconds [104]. In Figure 4.12 we show an extract of the results obtained by Laurent Pigeon during his thesis containing the time to execute the three most costly phases of the static scheduling.

Although the target application is different from ours, many of these concepts can be exploited for interactive simulations. Firstly, in a shared memory machine the deployment cost can be drastically reduced, since there is no network cost to be payed to transfer tasks, i.e. in shared memory machines assigning a K-Thread to a processor can be done by inserting a pointer in a list. The generation and partitioning time can also be reduced by simplifying these phases for a shared memory context as we show in Section 6.4.

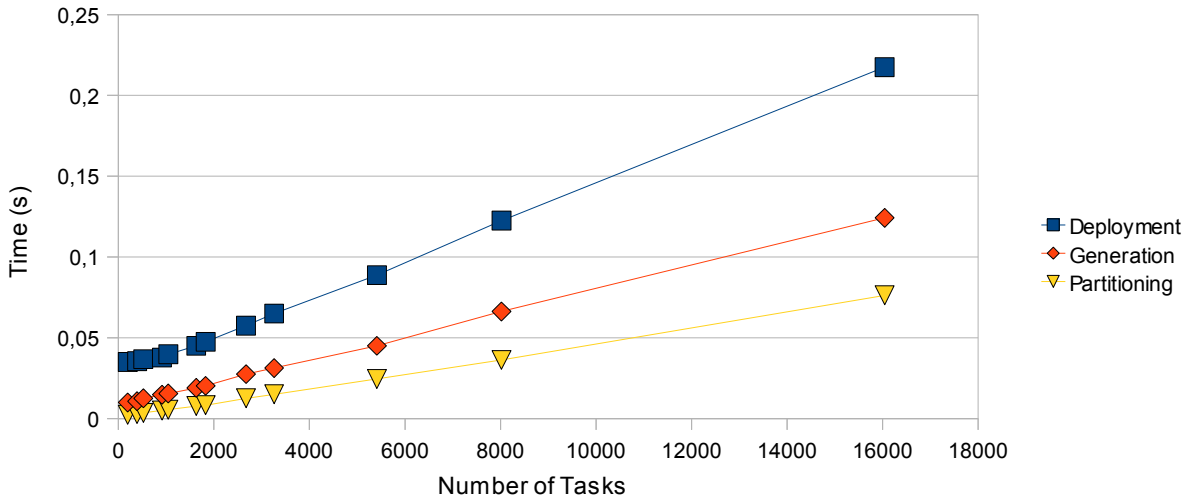


Figure 4.12: Static scheduling execution time. Partitioning enrich the task graph to ensure the communication between partitions. Generation groups the tasks from a partition in a single K-Thread. Deployment: deploys the tasks and data on all the processors

4.4 Parallel Physical Simulators

The choice of the parallelization approach can be influenced by diverse factors such as the target architecture, the original application and even the effort or budget allocated to the parallelization. At a coarse level we can find simple approaches, where different engines are executed in parallel, exploiting the **functional parallelism**. Looking inside a specific component, like a physical simulator, we can parallelize the operations of non-interacting bodies by executing distinct **collision groups** in parallel. Using a deeper evaluation we can exploit the parallelism at the **object level** by executing operations in parallel even when objects are colliding. Finally we can internally parallelize an arbitrary body using **data parallelism**.

4.4.1 Functional Parallelism

The basic approach for virtual reality application such as games is to split the application into separate components that communicate in a way that the output of one part serves as the input of the other. For instance it can lead to the execution of the physics engine concurrently with the artificial intelligence [112] and the visualization. This kind of approach reflects the structure of an application, but the amount of parallelism extracted is very limited and does not scale well. In most cases there is a restricted number of independent components, which are static regardless of the underlying architecture. On the other side, functional parallelism is much simpler to set up since independent blocs can be easily identified.

In most cases a simple data forwarding is enough for controlling the data flow between components. However more complex scenarios on virtual reality, such as immersion environments, require more sophisticated communications scheme. For instance the input frequency for a haptics device is much higher than the input needed by the visualization, which means that these modules cannot be fed with data in the same way. OpenMask [4], Avocado [124], Vista [114] and FlowVR [9] are examples of middlewares designed to couple independent modules to define a single virtual reality application

4.4.2 Collision Groups Parallelism

One example of collision group parallelism is a parallel version of the Open Dynamics Engine (ODE) [133] exploit parallelism between non-colliding bodies. It groups colliding bodies in sets, called islands, and creates one thread per island. Because every island is independent of one another, they can be computed in parallel with no shared data. Only the final body positions are sent back to a central thread at the end of each simulation step to update visualization and collision detection.

This parallel extension of ODE targets rigid body simulations. Rigid bodies physics are fast to compute, and the sequential execution of a small group of rigid bodies does not affect the interactive refresh rate. In addition, we can simulate hundreds of rigid bodies in an interactive rate using current computers, which increases the probability of having a large number of collision islands that can be computed in parallel.

This parallelization is efficient while the number of independent islands is enough to keep all the processors busy. As soon as there are less islands than processors the exposed parallelism is not enough to exploit all the available resources.

In terms of transparent parallelization, this approach meets our needs, since the physics code executed by each thread is the same as used in the non-parallel version. It means that, if new simulation algorithms are integrated in the simulator, they can be executed in parallel without additional effort. However, if we consider soft bodies, the drawbacks are more noticeable. Soft bodies are much harder to compute, and the time to sequentially compute a group of colliding soft bodies easily violates the interactive time constraints. In medical simulations, for instance, organs are placed in a constrained space and are colliding most of the time. Such a scene represents the worst case for this parallelization approach. If all the bodies are colliding, the collision detection algorithm will create a single island that groups them all. In this case, only one thread is used to sequentially compute the time step.

4.4.3 Body Level Parallelism

Body level parallelism computes independent functions in different bodies in parallel. To detect when two functions are independent it usually relies on temporal precedence constraints, or data dependency analysis. This deeper evaluation exposes more parallelism compared to the collision group approach. The body level parallelism relies on the fact that only a small part of the function that simulates colliding objects share the same data. All the remaining functions act on data that are internal to each object and can be computed in parallel. For instance, functions that update the internal state of the object, such as computing the new positions from the velocity can be executed in parallel on different objects even if they are colliding.

4.4.3.1 Temporal Precedence

Parallelism using temporal precedence can be obtained using **discrete event simulation**. In this approach the system behavior is modeled as a set of events that are chronologically ordered. An event can be described as a tuple, containing an action, and a date in the simulation time, so that an event changes the state of a system at a given moment in the simulation time. Additionally, an event can create other events to be executed later. An example of physical simulation modeling with discrete events is to use events to execute the operations of the time integration solver. The events can change the state of the simulated bodies, updating their position, and other internal data. Collision response can also be modeled as events that act on the colliding bodies correcting their position.

POSE [131] is a parallel environment targeted on discrete event simulation. It uses Charm++ [79] as a parallel environment. The discrete events are executed by POSE in an optimistic way. POSE speculates the execution of some events allowing different threads to advance computation as far as possible. In case of conflicts it must roll back the object states the last known stable state, and relaunch the events.

The roll back overhead is minimized using an adaptive speculation window. Objects that are historically successful on speculating events are allowed to go further, while objects having a high score of roll backs have a limited number of speculative events. Going further in the execution of one object before switching to another one favors cache reuse, resulting on better locality exploitation. Additionally this optimistic approach reduces synchronization between threads.

Brian Mirtich [90] proposed a rigid body simulation using discrete events. Similarly to POSE, it uses an optimistic algorithm based on timewarp [74]. The simulated bodies are considered independent from each other and can advance using large time steps. A roll back mechanism is used to treat collisions. If a collision is detected the state of the concerned objects is rolled back to the last known state where they are apart. The main advantage of this work is to allow an asynchronous simulation of bodies. Non colliding objects can be simulated using large time steps, and only the colliding ones must use smaller time steps to avoid interpenetration, which improves performance. This original solution does not exploit parallelism, but a similar approach could potentially be used on a parallel environment, for instance, by combining it with a parallel approach similar to what is done in POSE.

The discrete event simulation as presented in this section offers most of the functionality desired for interactive simulation. The avoidance of spurious synchronization and good data locality are fundamentals for good performance. However we want to go further and avoid spurious synchronization and rollbacking at the same time. Usually rollbacking is not a big penalty for rigid bodies. However, deformable bodies require storing a large amount of data, and undoing the changes in case of chronology violation adds an important overhead. Additionally, we must be sure that the object implementations have no side effects to rollback in case of conflicts. This constraint is hard to respect, specially if programmers are not aware of parallelism.

4.4.3.2 Data Flow Analysis

Another way to obtain a compliance with the sequential behavior is by using a data flow analysis. The analysis ensures that a task will start executing only after all its input data is produced. This information can be used by the underlying scheduler to choose which task to execute and also to improve data locality by executing together tasks that access the same data.

The data flow analysis is similar to what is done by some works presented in functional parallelism (Section 4.4.1). For instance, an extension of FlowVR simulates multibody physics using its functional data flow structure [10]. The virtual bodies are set as separate components, with external forces as inputs. Each component runs on its own process, computing its internal state and sending their position as output. Interaction components read the position of the objects to detect collisions. External forces are then applied to the colliding objects to correct their positions.

This distributed simulation shows how decoupled components can be combined in a high level way to obtain a parallel simulation. However this approach is limited to algorithms where bodies are weakly coupled. Once we use algorithms with strong coupling, it is harder to represent bodies as separate components, since it requires much more communication and the overhead added by the middlleware would not be negligible.

Charisma [70, 78], is an extension of the Charm++ environment. Like Charm, it decomposes the application in a set of communicating parallel objects. In addition, Charisma contains an abstract language that describes the global application algorithm, i.e the orchestration of the parallel objects. From the orchestration algorithm, Charisma connects the various components according to the control flow and data dependency analysis. Examples of charisma usage contains the description of the global algorithm of the NAMD molecular dynamics simulation [29, 83], developed using Charm++.

4.4.4 Data Parallel Physical Simulation

Fine grain data parallelism is the most common approach in soft body simulation. The gain in performance is obtained by decomposing the data domain and executing the independent parts in parallel. In contrast to the previously mentioned parallel simulations, the physics algorithms are internally adapted to extract data parallel zones.

An important point in data parallelism is data partitioning. A well-balanced work distribution across processing units is crucial to obtain good performance gain. Also the data distribution influences the border zone computation. If the coupling between parts requires too much communication it can add an important overhead. Depending on the parallel architecture it can result on transferring data over the network or can infer in memory contention on multi-core machines. Communication can also increase synchronization between processors, since one must wait from data to be produced by one another, limiting the performance of the parallel algorithm.

Most traditional scientific physical simulation methods focus on optimally partitioning complex objects, such as an atmospheric model for weather forecast [117], or interacting media in multiphysics applications [89]. We can find data parallel simulation in Clusters of PCs, for instance for cloth simulation using Athapascan [134], where parts of a tissue are distributed over the cluster nodes. In NAMD [29, 83], a molecular dynamics simulator written using Charm++ [79], the particles are divided in sub-domains and computed in parallel on a distributed memory architecture using message passing. On multi-core architectures we can also find cloth simulation [62] and finite elements simulation [75] using OpenMP. A fork and join approach can be found in cloth simulation using an extension of Cilk [123], or in a parallelization fluid simulation, and cloth simulations using Cilk, OpenMP and Intel TBB [71]. On GPU a wide variety of algorithms has been parallelized since data parallelism is intrinsic to this architecture. In this group we can mention meshless simulations [47] using CUDA and mass-spring model [56, 120] using shaders on a standard graphics pipeline. We can also find examples that combine one CPU with

one GPU for physics in games [77].

Some physical engines like PhysX [3] and Bullet [1] also defer computations to a GPU or SPU co-processor using data parallelism. It is not only used for time integration but also on collision detection. Such fine grain parallelization can be very efficient but requires a rewriting of each algorithm, which impairs productivity. In any case, a data parallel approach requires a strong knowledge of the algorithm being parallelized, since the programmer need to reorganize the code, or even propose alternative algorithms that better exploit parallelism.

4.5 Summary

In physical simulations, we can extract parallelism from different levels. At a functional level we can execute the visualization in parallel to the time integration (functional parallelism). Inside the time integration step we can compute the state of different physical bodies in parallel (task parallelism). And inside a given object we can update different elements of a vector in parallel (data parallelism).

In functional parallelism the amount of parallel work is limited by the number of the independent modules, which can lead to scalability problems. In data parallelism we can obtain a much larger amount of parallelism. However it usually requires rewriting the internal physics algorithms. Using a task level parallelism, we can exploit parallel architectures by executing operations in different objects in parallel without touching the internals of the physics algorithm implementations.

Depending on the available architecture, it is possible to take advantage of different levels of parallelism to improve performance. For instance, SIMD architectures are almost exclusively used to data parallelism, while MIMD machines are more flexible and can be used for functional, task, and data parallelism. For this reason, SIMD architectures will be considered later in this work when treating the specific case of graphics coprocessors. Our initial attentions are narrowed to task parallelism on MIMD machines, so that a larger number of non-experts in parallelism can benefit from the improvements obtained by our transparent parallelization.

Regarding the memory architecture, the main programming models for MIMD machines are message passing and shared memory. Message passing is often used for distributed memory machines, where the only way to share data is through explicit communications. This model has the advantage of being easily extensible, since new nodes can be added to the application if more computational power is needed. Also the explicit communication programming tends to make the developers write more structured code by keeping in mind all the communication costs. However parallelizing existing code using message passing usually requires restructuring the code or even completely rewrite. Which impairs productivity. On the other side, using shared memory model the programmer can access data from another processor in the same way as a local variable, which tends to be easier from the point of view of a non expert programmer. Sharing data between processor can be as simple as sharing a pointer to the memory which reduces the middleware software overhead.

Different parallel environments can be used to develop applications shared memory machines such as OpenMP, Cilk and Intel TBB. All of them extend the existing programming language by adding clauses to identify the regions of the code that can run in parallel. OpenMP support a fork and join approach, while Cilk and Intel TBB offers a task based approach for recursive parallelism with work stealing load balancing. The three environment also offer ways of doing data parallelization by means of parallel

loops.

The Atapascan/KAAPI programming environment offers a similar functionality for recursive parallel applications, with the advantage of expressing the dependency between data through a Data Flow Graph (DFG). Without this graph the programmer is forced to insert synchronizations to ensure a proper task executing ordering. On KAAPI we can maximize parallelism by avoid such spurious synchronizations. Like for Cilk and TBB, load balancing is achieved by a work stealing strategy.

Additionally KAAPI offers a static scheduling. It extends the applicative domain to non recursive applications, i.e., iterative algorithms. The DAG is described using the same API of the recursive approach, but instead of having tasks that create subtasks, all the tasks in the static graph are leaves tasks that only perform computations, without creating sub-tasks. A static scheduler evaluates the data dependencies and creates partitions that are distributed across the processors. Those partitions are completely autonomous from one another, which guarantees the best performances as there is no central routine controller.

Parallelization using graphics processing units drew a lot of attention for physics, first because we can expect these co-processors to be easily available on users machines, but also as it can lead to impressive speed-ups. The Bullet and PhysX physics engines defer solid and articulated objects simulation on GPU or Cell for instance. All these GPU approaches are however limited to one CPU and one GPU or co-processor. Task distribution between the processor and the co-processor is statically defined by the developer.

Recent works propose a more transparent support of heterogeneous architectures mixing CPUs and GPUs. GPU codelets are either automatically extracted from an existing code or manually inserted by the programmer for more complex tasks. StarPU supports an heterogeneous scheduling on multiples CPUs and GPUs with a software cache to improve CPU/GPU memory transfers. StarSs proposes three level scheduling composed by a master/helper/worker scheme. RenderAnts uses work stealing on multiple GPUs. Both RenderAnts and StarSs address multi GPU scheduling, but none include multiple CPUs.

Chapter 5

Task-Based Time Integration Parallelism

The works presented in this Chapter were published on [67] and is part of the scheduling strategies of KAAPI [2].

The development of advanced physical simulations, such as surgery simulations, requires the collaboration of specialists in various fields such as mechanics of continuous media, collision detection, numerical methods, geometry, haptics. In practice, most of them did not learn parallel programming. These specialists also need to easily experiment various models and algorithms, combined with each other's contributions. It is the case of SOFA, an open framework for physics simulation. To satisfy the requirements of collaborative developers in this context, parallelism should have the following properties:

- Non-invasive: The constructs used to express parallelism, i.e. to define task boundaries and data dependencies, should have a reduced impact on the code. It makes it friendly to algorithm developers and application developers who are usually not experts in parallel computing.
- Compatible: The parallelization should be generic enough to not impair further lower level parallelizations. For instance it should be possible to rewrite one specific algorithm to defer part of the work load on a GPU while keeping the benefit of the current parallelization. It makes it complementary with the popular GPU-based parallelizations.
- Externalized: The scheduling and mapping of tasks on processors should not be embedded in the physical simulation code. It eases code development, the programmer not being concerned about these aspects. It also enables to evaluate different scheduling strategies without code modifications.

To achieve these objectives we rely on task parallelism using the Athapascan interface and the KAAPI middleware. As discussed in previous Chapter, task parallelism applied to physics simulation makes it possible to execute operation on different bodies in parallel without the need of changing the internals of each physics algorithm. With task parallelism combined to the high level Athapascan interface we obtain a *non-invasive* parallelism. Also as stated previously, KAAPI supports dynamic load balancing by work stealing, which compensates the fact that a specific algorithm can be accelerated using fine grain parallelization, enabling a *compatible* parallelism. Finally by establishing a well-defined interface between physics algorithm and KAAPI, we *externalize* parallelism. With physics simulation code isolated from the parallelism aspects, developers can concentrate on physics algorithms. On the other side

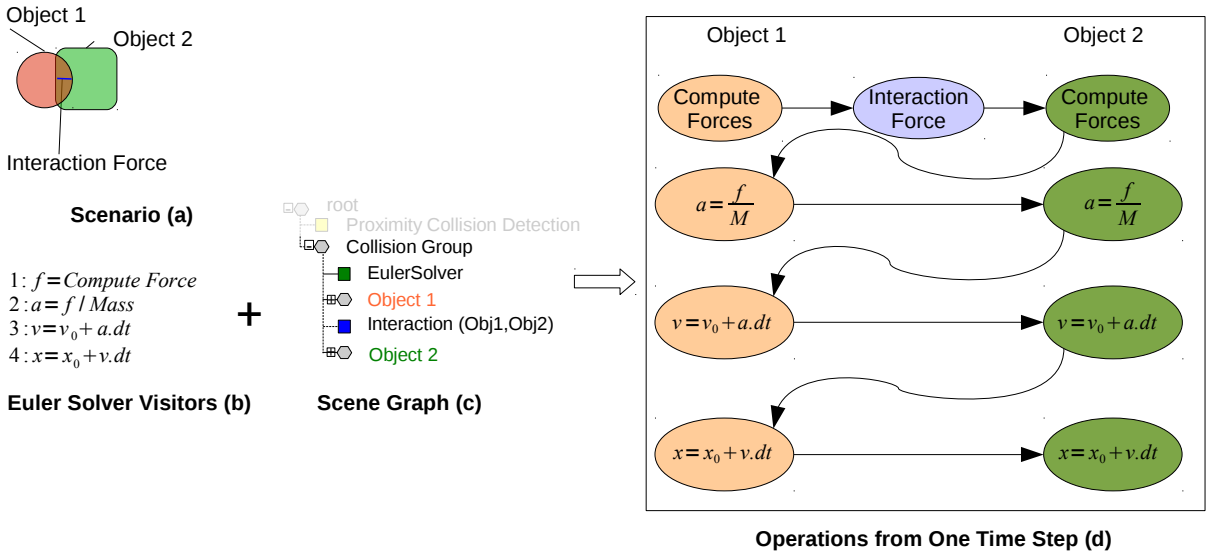


Figure 5.1: Time integration visitors execution: the Euler Solver Visitors (b) traverse the Scene Graph(c) triggering operation in a sequential order (d)

researches in parallelism can use these algorithms as a testbed and benchmark application, without the need to understand the physical behavior of the objects.

In this chapter we will treat physics simulation on multi-core architecture, so that we can concentrate in the non-invasive and externalized aspects of our work. We will study the requirements for interactive physics simulation, and well establish the interface between physics and the parallel part. In the next chapter we will discuss important points to improve the performance of parallel physics simulation, and also specific strategies to harness GPU and multi-core processors to validate the compatibility aspect of our approach.

We focus in the parallelization of the time integration step. However, collision detection can account for a significant part of the overall time. Parallel algorithms have been developed by reworking sequential ones like the recursive coordinate bisection algorithm [36] or regular voxel-based approaches [86]. Also in SOFA, a very efficient image based method has been developed [48].

5.1 Sofa Runtime

In SOFA, the simulation is represented as a scene graph that describes the shape and behavior of the different objects (see Section 2.5.2). The scene is animated by visiting each node of the graph and applying specific operations at each component.

For instance the visitors triggered by an explicit Euler time integration are listed in Figure 5.1(b). When traversing the scene-graph these visitors execute operations on each object (Figure 5.1(d)), computing the new velocity from the current acceleration, or the new position from the velocity.

5.1.1 Visitor Level Parallelism

In Chapter 2 we have seen the different steps that compose a collision detection pipeline. It creates and removes contacts based on geometry intersections. These changes are reflected to the scene graph by adding and removing components and nodes Additionally, the colliding objects are grouped in a same sub-tree (Figure 5.1(c)).

One way to exploit collision groups is to use the same instance of the time integration solver (Euler Solver, for instance) for all the objects in the same collision group. This approach, sometimes called *strong coupling*, improves the simulation stability (see Section 2.4.3). Objects that are in different groups does not interact with each other, consequently each group can be processed independently from the others.

A first SOFA parallelization attempt takes advantage of this intrinsic independence between collision groups. It offers a level of parallelism similar to the related works presented in Section 4.4.2. This parallel version was implemented with KAAPI and Cilk, using the hierarchical structure of the scene-graph to spawn tasks recursively.

During the traversal of the scene graph, the visitors spawns tasks that are charged to visit a branch of the scene graph. These tasks can spawn other tasks when a new branch is found. The visiting process of different branches is performed in parallel, allowing to parallelize independent objects.

When considering colliding objects the level of parallelism is much more restrained. Due to the sequential aspect of the high-level algorithms, synchronizations happen at the end of each visit to a sub-tree. This synchronization is needed to make sure that the data used by the next visitor are available. As illustrated in Figure 5.2(a), the synchronization reduces the parallelism of a time step, which dramatically decreases the speedup (performance results are presented farther in Figure 5.9.

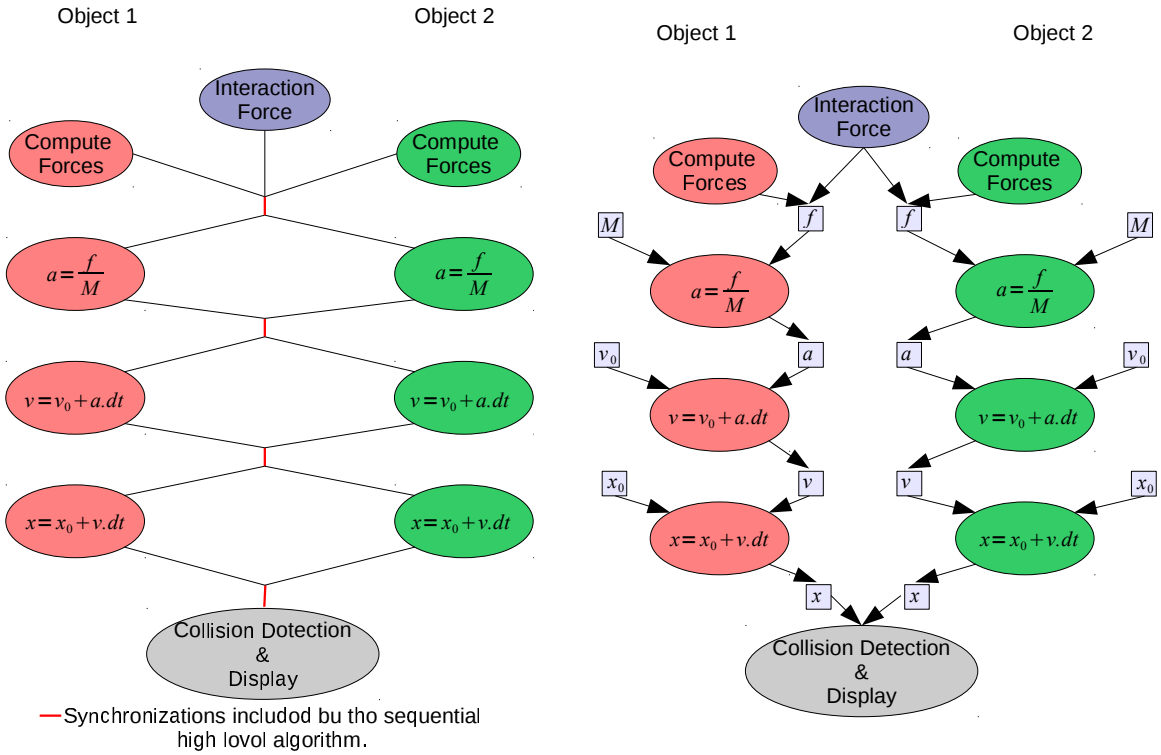
5.1.1.1 Component Level Parallelism

To better exploit the parallelism inside a collision group we relied on data dependency analysis. Using the Athapascan middleware [55], we instrument the code of the SOFA framework to identify tasks as well as the data shared between them.

The data are vectors representing the physical states of an object, like positions, velocities and forces. The tasks are operations on those vectors like accumulating forces, computing positions from velocities, etc. The graph edges represent dependencies between tasks and data. Figure 5.2(b) corresponds to a simplified graph of one explicit time integration step where two objects are colliding.

We produce data flow graphs corresponding to the different operations triggered by the equation solvers, rather than directly performing these operations. Each visitor fired by the algorithm produces tasks, and the complete sequence of operations performed by the algorithm is represented as an assembled graph that precisely models data dependencies at component granularity. With a good partitioning of this graph we can avoid the undesirable synchronizations previously discussed, and significantly improve performance.

Let us consider the example of a simplified SOFA scene like shown in Figure 5.1. The execution of this simulation using a recursive approach results on an execution sequence that is over synchronized



(a) Recursive

(b) Data Flow Graph

Figure 5.2: Parallel physical simulation using a recursive approach and a data flow graph.

(Figure 5.2(a)). However, when using KAAPI with a static scheduling, we are able to fully exploit the task parallelism of this scene. To obtain a similar result using the aforementioned programming environment, we would need to reorganize the code to explicitly express that there is no data dependency between the tasks associated with different objects.

The solution based on static scheduling is inspired by the works of Laurent Pigeon [104]. These works as well as some aspects of KAAPI that are linked to this thesis are described in Section 4.3.

5.2 Transparent Parallelization

The new task parallelism interface presented in this chapter was employed to parallelize SOFA. In SOFA, each type of component has a well-defined interface, which specifies the operations that should be implemented, together with the external data available to this operation. The function defined by the framework are then called by the visitors. For example, a time integration solver triggers a visitor to compute the forces. This visitor will then call the associated function on each component. In the case of a Force Field, it will call the *addForce()* member function (Figure 5.3). The *addForce()* obtains the references to the data, and calls the component implementation (Figures 5.4(a) and 5.4(b)).

```
class ForceField{
   ...
   virtual void addForce(Vector force,
                         Vector position,
                         Vector velocity)=0;

   virtual void addForce(){
      addForce(getF(),getX(),getV());
   }
      ...
};
```

Figure 5.3: Framework Interface

```
class SpringForceField: public ForceField{
   ...
   virtual void addForce(Vector force,
                         Vector position,
                         Vector velocity){
      ...
      /* addForce implementation
         for a Spring          */
      ...
   }

   ...
};
```

(a) Spring Component

```
class HexahedronForceField: public ↩
    ForceField{
   ...
   virtual void addForce(Vector force,
                         Vector position,
                         Vector velocity){
      ...
      /* addForce implementation
         for a FEM             */
      ...
   }

   ...
};
```

(b) Finite Elements Component

Figure 5.4: Component specific implementation for the addForce function

To transparently parallelize these functions, we straightforwardly turn the elementary operations of the framework into KAAPI tasks. As shown in Figure 5.5, the task declaration also contains the data

```
struct ParallelAddForce{
   void operator()(ForceField *ff,
                   Shared_rw<Vector> force,
                   Shared_r<Vector> position,
                   Shared_r<Vector> velocity){
      ff->addForce(force,position,velocity);
   }
};

class ForceField{
   ...
   virtual void addForce(Vector force,
                         Vector position,
                         Vector velocity)=0;

   virtual void addForce(){
        Fork<ParallelAddForce>()(this,getF(),getX(),getV());
   }
   ...
};
```

Figure 5.5: Framework side changes to support task level parallelism

access pattern for each parameter. For example the *force* is accessed in read-write mode, while position and velocity are read-only variables.

When a SOFA visitor calls a *Fork* it does not execute the tasks. It only adds the task to the data flow graph. The task will be executed only after being scheduled and assigned to a processor. At this moment, the KAAPI task calls the virtual methods overloaded by the components. This execution pipeline is depicted in Figure 5.6, where we show the software coupling between KAAPI and SOFA.

Using this software structure, the experts in various disciplines who collaborate on the development of the library can safely ignore parallelism issues if they respect the interface and only access components data through this interface. In other words, all the shared data needed by the function must be part of the function parameters. By doing so, the component implementation can then be considered as a black box by the execution environment, allowing the developer to code it sequentially.

At the same time that it allows to develop sequential internal code, this does not prevent a fine grain parallelization at a data level, by deferring computations to a co-processor for instance. The dynamic load balancing will ensure a good exploitation of the resources even with multi-grain parallelism and heterogeneous architectures, scenarios where it is hard to predict a good static partitioning.

5.3 Graph Partitioning and Mapping

Once we have a task graph representing the simulation operations, we can schedule this graph by assigning the tasks to a set of processors using a scheduling algorithm. In most of the cases we can deduce the task placement from the scene structure. In Figure 5.2(b) all the tasks after the force computation modify the data of only one body. In this case the tasks can automatically be gathered in the same partition accordingly to the body it modifies. The remaining tasks that are associated to multiple bodies, like the *interaction force*, are grouped with other tasks that modify the same set of bodies.

If needed, we can also employ a dedicated partitioner like SCOTCH [103] or METIS [81], to better

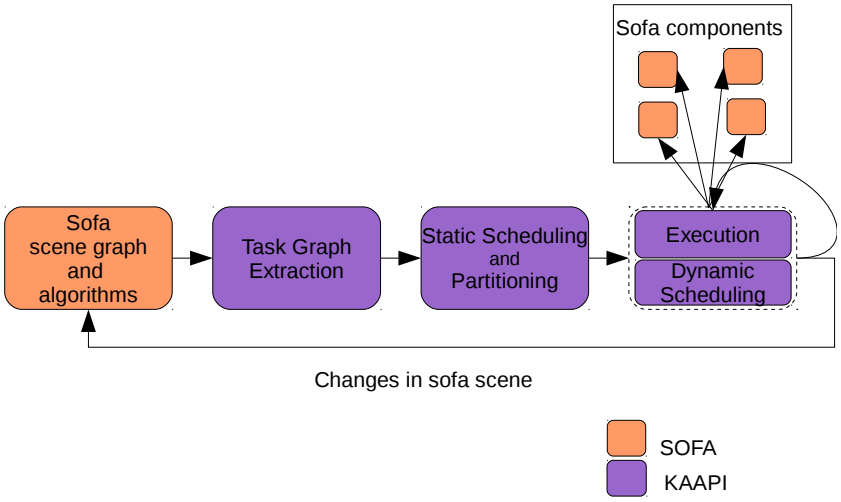


Figure 5.6: Execution flow coupling SOFA and KAAPI. SOFA generates a task graph using the KAAPI interface. This graph is partitioned and execute. During execution the tasks call the SOFA components.

optimize the scheduling. These libraries offer a set of algorithms to partition a graph following different criteria. For example we can ask for a scheduling that minimizes the communication between partitions. One situation that may need a deeper analysis of the dependency graph is for extracting parallelism from the tasks related to the same object. A dedicated partitioner could be able to extract some internal parallelism, while using our previously explained approach all the tasks from an object are gathered in the same partition, and consequently executed sequentially.

On the other hand, these partitioners are time consuming, and the time spent on scheduling can be unsuitable for interactive simulations. Additionally, an external partitioner follows a partitioning criteria given by the user, which has no guarantee on the relationship between the scene and the partitions. The result is strongly dependent on the scheduling criteria. If too many partitions are requested, the scheduler will probably separate tasks from the same object in different partitions. In the opposite way, when asking for fewer partitions than objects, we will have tasks from different objects placed in the same partition. It tends to have no drawback when we create as many partition as processors, and the load remains well-balanced during all the execution. However re-balancing the load in these cases is much harder, usually requiring to repartition the graph.

Gathering in the same partition the tasks that access the same data (Owner Compute Rule) leads to a partitioning that corresponds to the scene structure. It enforces data locality, as the data and all tasks accessing these data can be gathered in the same processor. If there are enough partitions the bodies can be easily re-balanced over the processors without requiring to repartition the graph. A a computationally expensive object can migrate from an overloaded processor to an idle one only by reassigning the partition related to this object.

5.4 Cumulative Write

From the scheduled task graph, we create the K-Threads that will be executed in parallel. When one task on a K-Threads depends on data from another K-Threads, we need to add special tasks that are charged to synchronize both K-Threads. The strategy employed is the same as proposed in Section 4.3.4, where a *signal* task is placed in the writer thread, and a *wait* task is placed in the receiver thread (Figure 4.11). These controlling tasks are useful when we have a writer/reader pattern.

However, most of the communications between K-Threads can be expressed as cumulative operations. Considering the case where two objects are colliding. The K-Threads need to communicate to apply the forces on each object. In this case, forces are accumulated on each object to separate them (Figure 5.2(b)). Cumulative operations can also be found in the conjugate gradient algorithm when computing a dot product. This kind of operation is commutative as each writer increments the variable by an arbitrary value, and the order that these accumulations are done does not changes the final result. If we isolate the contribution of each writing task we can execute them in parallel. Once the tasks are finished we can reconstruct the final result by summing all the contributions.

In the Athapascan interface, we can declare a parameter as cumulative write using the `shared_cw` keyword. To implement a parallel cumulative write, we decompose it in two phases: accumulation and reduction. At the beginning each writer accumulates its value to a temporary buffer, so that they can run without concurrency issues (Figure 5.7). The *reducer* waits for all writers and adds the value of all the accumulated temporary buffers to the previous value. From the point of view of the readers, the data are considered produced only after the *reducer* task has finished. Even if theoretically they are writers of the data, in practice they only write in temporary buffers and the *reducer* is the real writer of the data.

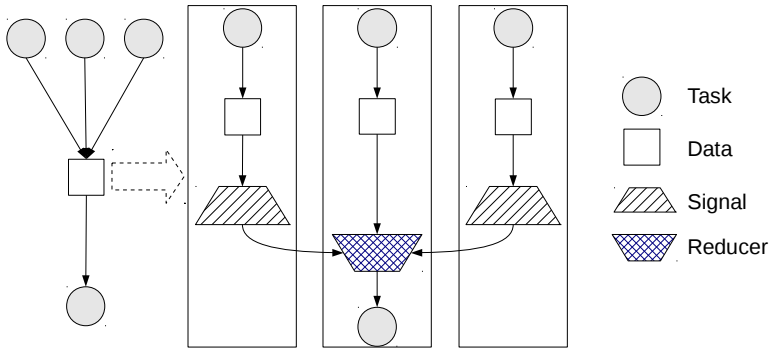


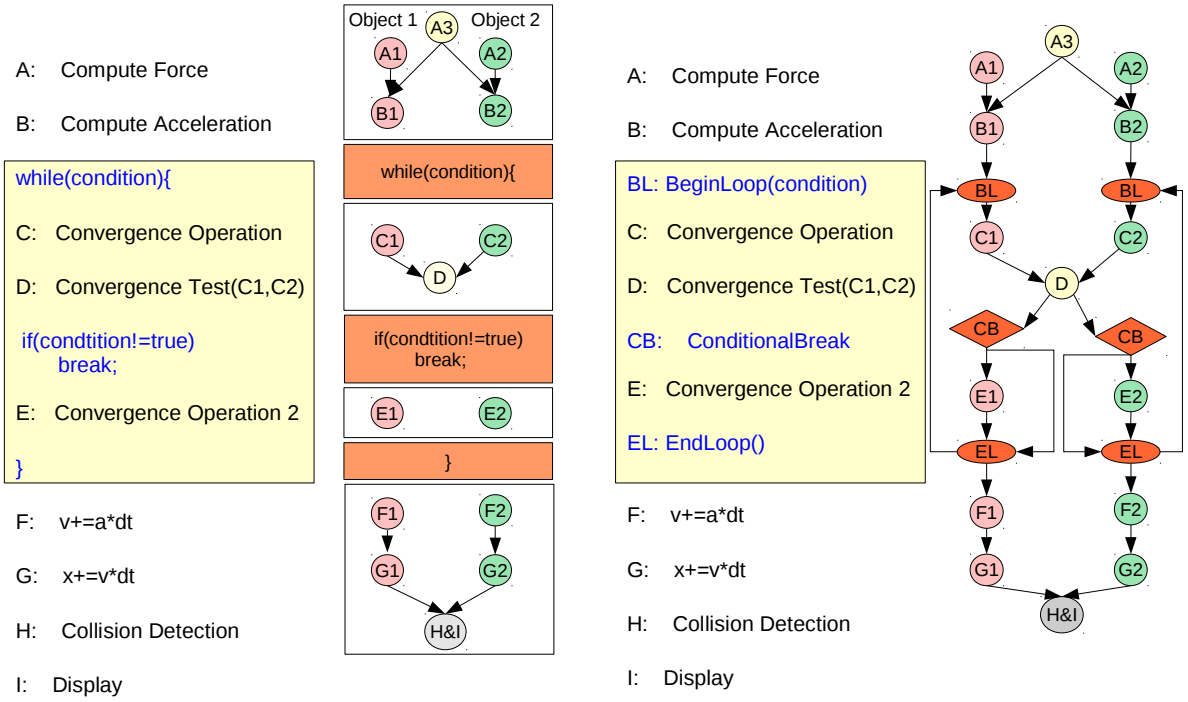
Figure 5.7: Control tasks employed to guarantee data access coherence between different partitions. Left: initial data-flow graph. Right: graph after partitioning. All the writes are performed in a temporary buffer that will be further accumulated by the Reducer.

This new way to control the parallel access to the data allows to expose more parallelism. Without this cumulative operation a cumulative data access would be considered as a read/write access. In this mode only a single writer is allowed to access the data at a time resulting in a serialized execution of the tasks. On the other side, when using a cumulative access, all the involved tasks can execute in parallel.

5.5 Dynamic Loops

As stated in Chapter 2, some integration processes. It is the case of the conjugate gradient that iterates up to comply with a given convergence criterion. There are different ways to exploit the parallelism of these iterative algorithms. When we have a fixed number of iterations that is given from the beginning we can unroll the loop and process it as a sequence of tasks. It requires no additional software structure to control the loop. On the other hand it limits the variety of supported algorithms. To support loops where the number of iteration is dynamic, we can employ the control structures of the programming language, as shown on Figure 5.8(a), and re-spawn all the tasks when a new iteration must be computed. Another solution is to identify the loop code, extract the tasks, partition it and redeploy it for each iteration.

The first approach is not suitable for physical simulation since some equation solvers have dynamic loops. The second solution offers dynamic loops, however the control structure is centralized on a single thread, which will create all the tasks to compute an iteration. It adds an extra synchronization barrier as the main thread must wait for all the threads to end before redeploying the next iteration.



(a) Dynamic loop using programming language structures

(b) Dynamic loop using Athapascan

Figure 5.8: Dynamic loop. Functions e1, e2, g1, g2 change multiple objects. Left: standard code with its execution flow. Right: Our modified code with the graph representation of our implementation.

We adopted the third solution where the loop is expressed in the task graph, partitioned and re-deployed at each iteration. Aiming to offer a programming interface that supports dynamic loops, we extended the Athapascan interface by introducing two special keywords, *EndLoop* and *BeginLoop*. These keywords create special tasks that are used to delimit the boundaries of a loop. All tasks created between the *BeginLoop* and the *EndLoop* are considered part of the loop body. This loop is controlled through a condition variable or a static iteration counter. We also introduce a *ConditionalBreak* construct that can

be used to test the conditional variable inside the body of the loop. This break instruction can be very convenient to exit the loop from any place in the code.

To extract parallelism from such a loop, we need to flag the loop boundaries and conditional breaks so they can be identified in the task graph. It is done by representing them as tasks in the graph. In terms of data dependencies, these tasks access the condition variable as readers, and then are dependent from the tasks that write to the condition variable. The scheduling of the tasks inside a loop takes the same assignment pattern that is used for other tasks as explained in Section 5.3.

More particular treatment is done during the partition generation, where we create the sets of independent threads. If a loop is decomposed into several partitions, the loop control tasks are replicated on all these partitions. It ensures that all the partitions will leave the loop at the same execution level. The break condition can be a boolean value that is shared by all the partitions. Also the condition can be the number of loop iterations. The main difference is that using a condition variable that is shared by all the partitions, we have a synchronization barrier as all the partitions must wait for the task producing the condition variable. In the case of a loop controlled by a fixed number of iterations the iteration counter of each partition can be incremented locally. It allows to asynchronously execute loops from different partitions, as far as there is no data dependency between them.

The condition is tested each time the *BeginLoop*, *EndLoop* or *ConditionalBreak* tasks are executed. When reaching a *EndLoop* task , all the tasks of the loop body are redeployed if the loop condition is still valid. Otherwise, we must exit the loop. The next task to be executed will be the task just after the *EndLoop*.

In Figure 5.8(b) we show a loop executed onto two partitions. The task *D* updates the condition variable at each iteration, and all the loops that depend on this variable are readers of this shared data. By considering a condition variable like any other shared data, we guarantee that the access control is made automatically by the *signal/wait* mechanism explained in the previous section.

Using this new mechanism allows to reduce the number of synchronizations. Unlike using the loop constructs of the programming language that requires a synchronization to evaluate the loop condition. In our case each *BeginLoop* task, tests the condition in parallel, allowing to start executing the next iteration without waiting for the other threads. Also the redeployment of the task inside a loop are done in parallel allowing to quickly start a new iteration by reusing the data structures from the previous one.

5.6 Partition Stealing and Execution

One of the advantages of static scheduling is to group related tasks in a same partition, and to create partitions that are weakly dependent so that we can better exploit the parallelism. In a pure static scheduling approach, a partition is assigned to a processor and remains there during the whole execution. This static assignment does not affect the performance if a well-balanced partitioning is provided. Different static scheduling algorithms intend to do so, but they are computational expensive. It is not a problem for applications like Computer Aided Process Engineering [22], where the cost of an iteration is much larger that the time to partition and schedule the application.

However, for interactive applications, the time to compute an iteration is much shorter, usually a few milliseconds to enable interaction. In Section 5.3 we presented a simplified scheduling algorithm

that relies on hints given by the application, for instance the relationship between a task and a simulated body. This direct assignment makes it possible to avoid the scheduling phase since the mapping between task and partition is given by the application. However this kind of scheduling can lead to unbalanced configuration, when for instance during the distribution of the partitions all the large bodies are assigned to the same processor.

By associating, static and dynamic scheduling, we can benefit from task partitioning and load balancing. The static scheduling explained previously in this chapter is used as an initial partitioning of the application. Once the execution starts, a work stealing approach is employed to dynamically redistribute the load.

Like usually done by KAAPI, the stealing process is decomposed in two phases : A partition (K-Thread) level stealing, and a task level stealing. In the first case a thief thread will visit all the processors trying to steal a ready to execute partition. If it fails to steal a partition, the idle processor will try to steal a ready to execute task, like explained in Section 4.3.3.

Giving priority to the partition stealing instead of task stealing has two main advantages. First we enforce data locality, since tasks accessing a same data set are in a same partition and when stolen they all move together to the stealer processor. Next we obtain a larger amount of work on each steal, which will result on fewer steals and consequently reduce the parallelization overhead.

5.7 Results

We tested our approach with different simulation scenarios, going from identical objects that are completely independent to heterogeneous scenes of colliding objects. The tests were performed on one computer equipped with 16 cores (8 dual-core 2.2GHz Opteron processors) with 32GB of RAM. Each processor has direct access to 4 GB of memory and uses Hypertransport to access the other processor memory, leading to non-uniform memory accesses (see Section 4.1.2) .

We only consider the performance of the time integration step. The speedup is computed taking the sequential execution time T_{Seq} as the reference. Task graph partitioning, that only occurs when new collisions are detected, is always included in the measured time.

5.7.1 Independent Similar Objects

The first test is a scene composed of non colliding identical objects simulated independently. Each object is a soft bar that is attached at one end and gets deformed due to the gravity force. This test highlights the overhead induced by the parallelization.

We compared our approach with two other implementations which we briefly describe here. The first one uses Cilk and relies on a purely dynamic load balancing scheme using work-stealing. We also compared to an implementation using recursive parallelism on KAAPI, following the Cilk divide and conquer approach.

A divide and conquer scheme can be obtained from the hierarchical representation of a scene in SOFA. The nodes of the scene graph are recursively visited to perform the operations defined by the

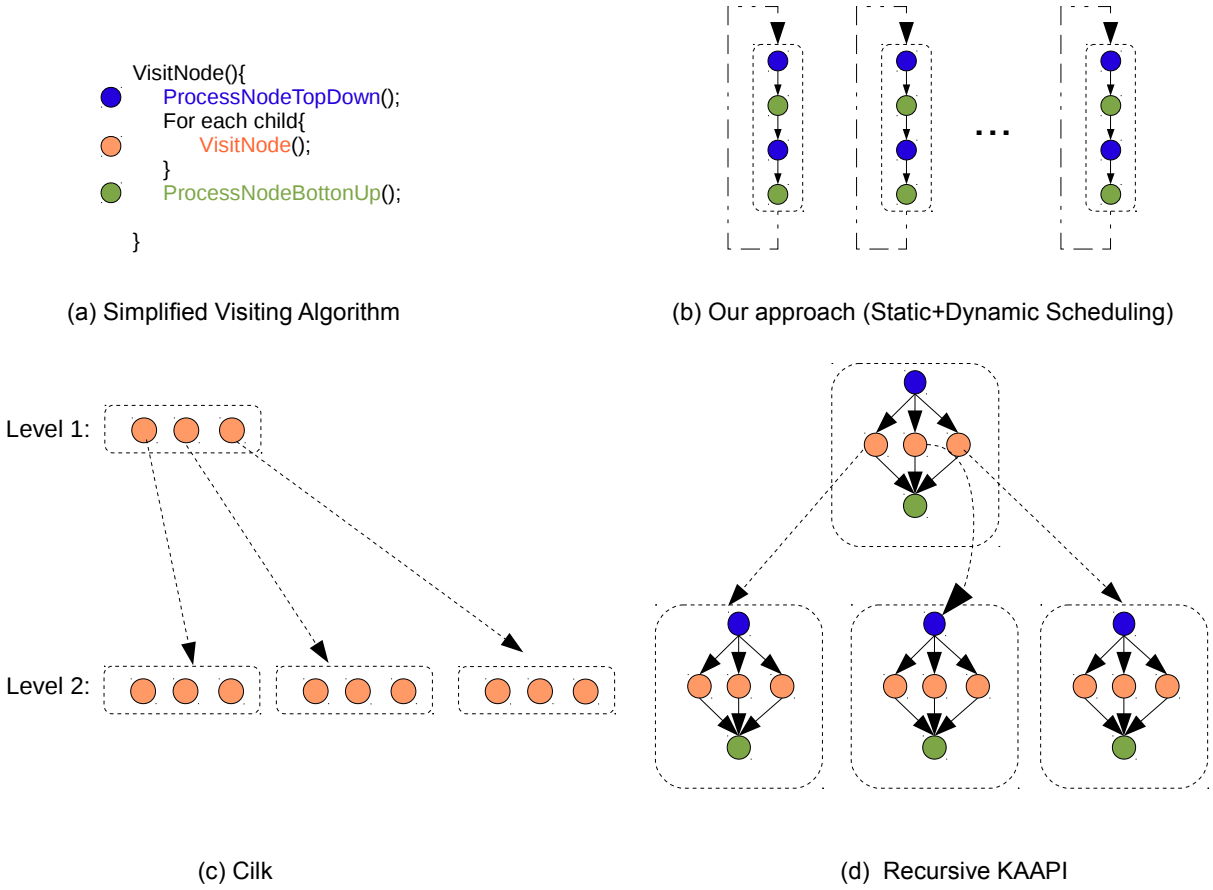


Figure 5.9: Task deployment structure.

high level algorithms. This traversal is performed using three steps like shown in Figure 5.9(a). First we execute the top-down operations on the nodes. Next we recursively visit all the child nodes. At last, once all children have finished, the visitor executes the bottom-up operations for the current node.

Each tool offers a different way to parallelize such scheme. Using Cilk, a task depends only from its relative parent. When two tasks at the same level depends on each other, we can use synchronization barriers to guarantee that all the preceding tasks have finished. It is the case of the VisitNode function. We cannot create the children visitor tasks before the top-down operation has finished, neither execute the bottom up operation before all the children had finished. In this case it becomes faster to sequentially execute the bottom-up and top-down function, instead of spawning a task and synchronizing just after. That is why we have implemented only the VisitNode function as a task, executing the other operations sequentially, which results on a spawning tree like shown in Figure 5.9(c).

Using KAAPI we can express the dependencies between tasks even when they are at the same recursive level. It allows us to create a top-down task and specify that all the visitNode tasks depend on it. Also we can create a bottom-up task that depends on all the proceeding visitNode tasks, the expressed data dependencies ensure a proper scheduling. It is done without using synchronization barriers. The resulting spawning tree is depicted on Figure 5.9(d).

The test runs with 64 bars, each one is composed by 256 particles that are simulated using hexahedral

finite elements (Figure 5.10) with an Implicit Euler solver. This size of the objects is large enough not to fit in cache, and at the same time small enough to avoid having the memory as a bottleneck. Since there is no collision in this scene, the task graph remains the same all over the simulation, i.e. it is partitioned only once at the beginning and redeployed all over the remaining iterations. Also as it is a homogenous scene, most of the time the processors are in a well-balanced configuration. The rare steals are a result of external interferences in the processors load.

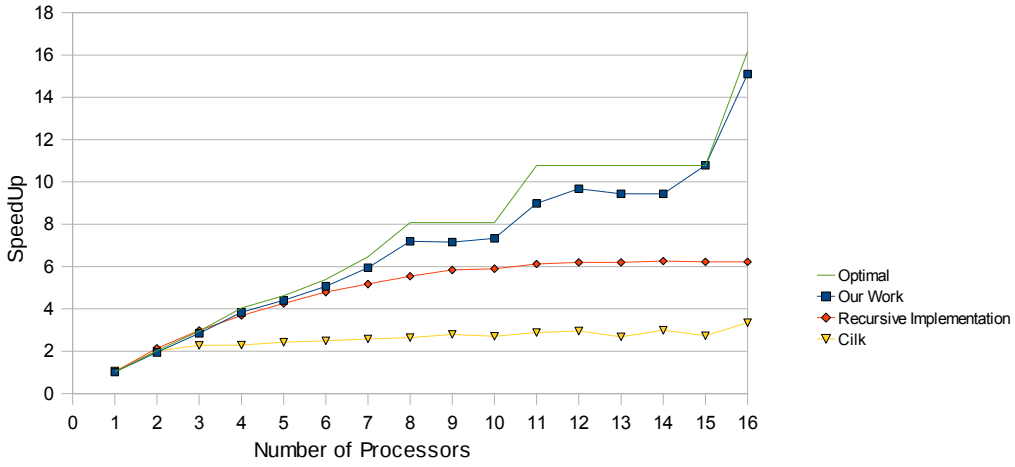


Figure 5.10: Speedup using 64 identical bars of size 16x4x4 without collision. $T_1 = 150ms$

Our implementation leads to a speedup of 15 with 16 cores. (see Figure 5.10). We succeeded to manage the most important issues that are not treated by previous work: spurious synchronization barriers and locality coherence over iteration steps.

The spurious synchronization barrier is a drawback of the recursive approach. A task is considered as finished only after all the children have finished. When we fire a visitor, there is only one initial task that recursively spawns other task as it traverse the graph. Since the high level algorithm is sequential and fires one visitor after the other, we need to wait the end of the previous visitor execution, before firing the next one. Using a static partitioning, we unroll this recursive execution, and evaluate the data dependency among the leafs of the graph. We can then avoid spurious synchronizations by linking tasks created by different visitors using the dependency between the data that they access.

In both previous works, Cilk and recursive KAAPI, there is no parallel dynamic loop. All the iterative algorithms were expressed using the programming language structures, resulting on a centralized loop control. Although the same operations are executed over different iterations, they are created as new tasks and have no link to the task executed at the previous iteration. It means that from one step to the other, there is no guarantee that a task will execute on the same processor. In our case, a partition always stay mapped to the processor that ended its execution in the previous iteration, which enforces data locality across iterations. Considering that the architecture used on this test has a non-uniform memory access, a good exploitation of data locality is crucial to obtain an efficient running.

To understand the differences between Cilk and KAAPI we need to analyze both spawning trees (Figures 5.9(c) and 5.9(d)).When a visitor is fired only one processor is active, all others have no work, and are trying to steal tasks. On Cilk we need to wait the end of the *ProcessNodeTopDown* function to

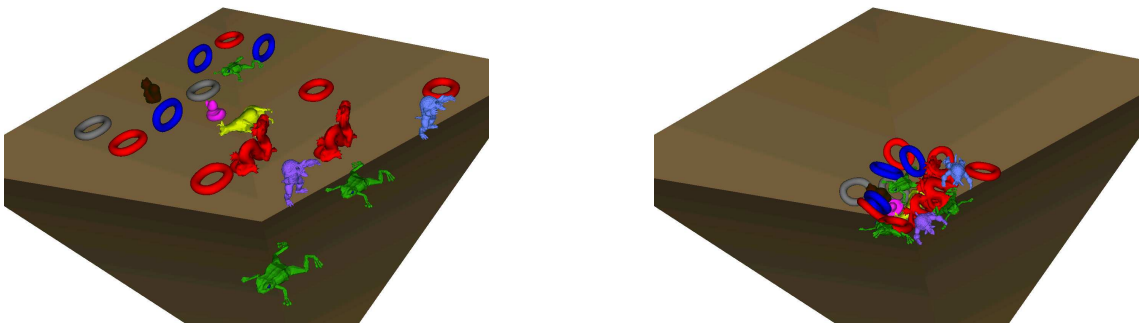
start spawning `VisitNode` tasks. It means that all the other processors remain idle up to the creation of the first `VisitNode` task. On KAAPI, it creates the *ProcessNodeTopDown* task, and then creates the `VisitNode` tasks specifying that they depends on the *ProcessNodeTopDown* task. Finally it creates the *ProcessNodeBottomUP* which depends on all the `VisitNode` tasks. It is only at this moment that the main thread will start executing the tasks that are created. However, in the meantime, the *ProcessNodeTopDown* was probably stolen and executed by an idle processor, and some of the `VisitNode` may have been already executed by other processors. Consequently, the remaining workload found by the main thread after spawning the `VisitNode` tasks is much smaller when compared to Cilk, which results on a reduced execution time.

Compared to the recursive paralllelization using KAAPI, our approach allows to obtain better performance since no spurious synchronization is inserted. Also we reuse the tasks from previous iterations to redeploy the next one, allowing to improve data locality by preserving the placement over different iterations.

5.7.2 Heterogeneous Scene

The parallelization of a scene composed of independent objects can be obtained using simple approaches, like simulating each object in a different thread or even in a different process. However when objects are colliding, this method can only take advantage of a reduced amount of parallelism. This difficulty of extracting parallelism is one of the reasons why some physics simulators consider a group of colliding objects as an indivisible component [133]. This is for this kind of simulation that our approach shows its full potential.

In Figure 5.11 we have the initial and final state of a scene of heterogeneous objects falling under gravity and colliding (see the video for a full simulation). The different objects have surface meshes that range from 400 to 2.500 triangles. The method employed to simulate the object varies: there are rigid bodies, and soft bodies using mass springs, finite elements or deformable grids models. Interaction between object is modeled using triangular meshed, and collision response is done by penalty using elastic stiff forces. For time integration we used an Implicit Euler method.



(a) Initial State (b) Final State

Figure 5.11: Scene used for the complex simulation tests. Objects are simulated using different methods, like Finite Elements, Springs and Regular Grids

The speedup evolution over time is shown on Figure 5.12. During the first steps we obtained the best speedup. At this phase there is almost no collision between objects, and as explained in Chapter 2, they can be solved by separate instances of the solvers. The best speedup in this initial phase is less than half of the optimal, since it is affected by the heterogeneity of the objects.

As we approach the final state, objects get closer, and at the end they are all part of one single collision group. The instance of the solver employed on all the objects is the same to avoid instabilities (*strong coupling*). It means that they all must share the same iterative loop and break conditions. In addition, due to the collisions, many tasks access data from other objects, creating synchronization points.

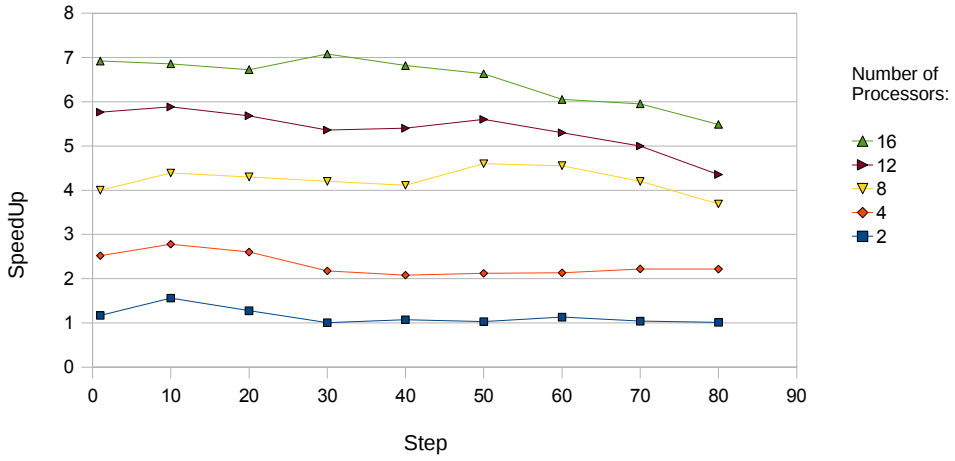


Figure 5.12: Speedup on a heterogeneous scene containing different objects simulated using different mechanical models.

When collision groups change, the task graph is updated. The cost of this procedure is proportional to the number of contacts in the scene. In the scene from Figure 5.11 the mean cost of updating the graph is close to 30ms, which is about the same time required to execute a timestep. This overhead is usually not noticeable (see the video). It is compensated by the overall performance improvement and is often amortized over a few timesteps.

The speedup obtained here did not require any effort from the application developer nor the physics algorithm developer. The physics algorithms can evolve independently from the parallel code, reducing the lag between the development of sequential algorithm and the execution of this algorithm in a parallel architecture.

5.7.3 Domain Decomposition

Our approach targets a coarse grain parallelism without looking at the internal implementation of a given method. In a scene with few objects or with a huge time-consuming object, it would be harder to obtain good performance gains. The exposed parallelism in such case is not enough to exploit the capacity of all the processors. In these cases a finer decomposition would help to obtain a larger number

of independent tasks.

SOFA enables to tightly connect different objects into a scene by means of constraints. The constraints can be used to ensure that two points from different objects will be considered as a single one during the simulation. For instance, when two objects are attached using constraints the forces applied to a given DOF in an object will also be applied to the corresponding DOF in the other object. Using constraints we can explicitly decompose a large object into smaller connected ones. SOFA guarantees that both physical simulation will be identical.

Each sub-part of an object corresponds to a different node in the scene graph. The constraints that attach them are components that modifies the state vectors of different objects. If we take the example of a scene with colliding object, a constraint between attached objects is the analogous of an interaction force between colliding objects.

A procedurally generated soft tissue was used to evaluate the domain decomposition. The original mesh is subdivided in horizontal stripes as shown in Figure 5.13. The mechanics of each sub-part is simulated using a mass-spring model, and a constraint is employed to attach the points in the border of each sub-part.

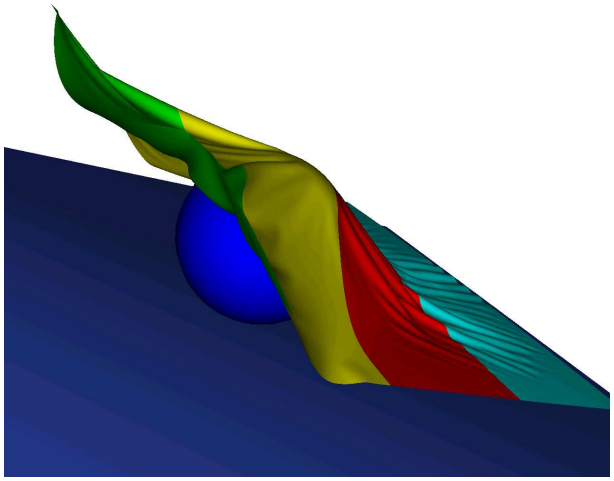


Figure 5.13: Domain Decomposition Scene: to expose more parallelism a soft tissue is subdivided in horizontal stripes.

Using this simple domain decomposition approach, we tested three different sizes of meshes. The results are shown in Figure 5.14. We obtained speedups of about eight in the best case with a minimal parallelization effort. We obtained better results with smaller objects as it can benefit of a better cache usage. As the object size grows the overhead introduced by memory access affects the performance.

Notice that all the parts share the same iterative loop for the convergence phase of the conjugate gradient. It means that they are solved using the same equation system. Parallelizing this scene without data dependency analysis would expose insufficient parallelism as there is a single high level algorithm controlling all the sub-objects, which is sequential.

Leaving the decomposition as a part of the scene description seems to be a good alternative, to avoid to have the domain decomposition internally coded in the physics algorithms. People who develop

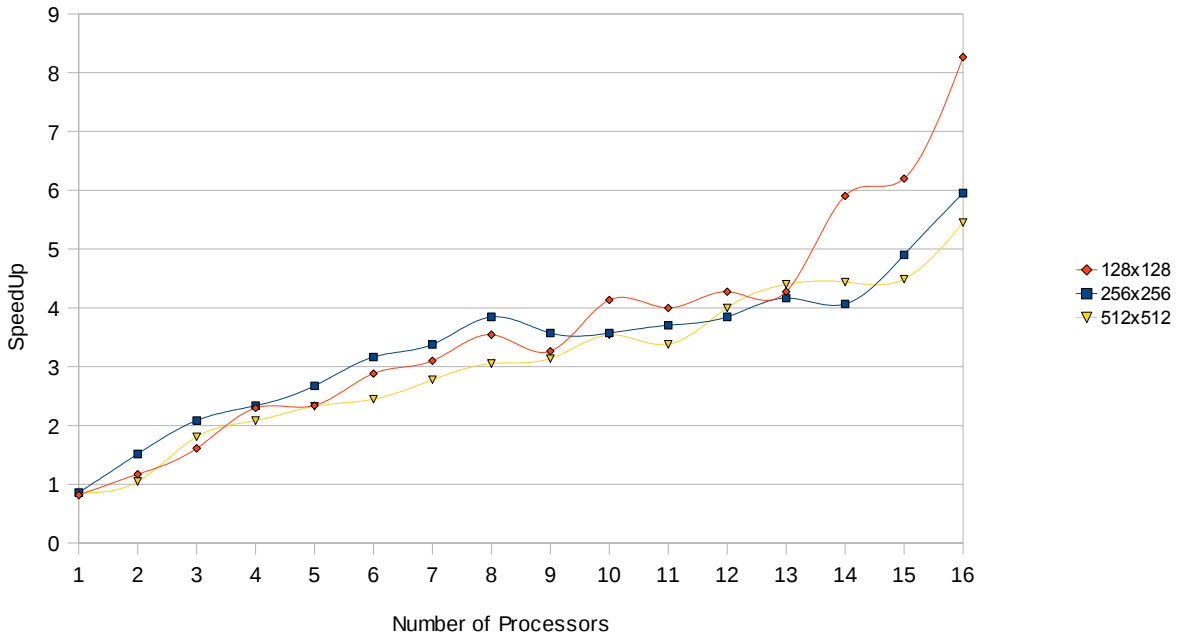


Figure 5.14: Speedup using a mass-spring mesh simulation by domain decomposition. The sequential time for each case is: 125ms (128x128), 500ms (258x258), 2.2s (512x512)

components can continue to code as if it was a sequential program. The one who wants to make a domain decomposition needs, at its turn, to set components that are charged to control the interaction between the different sub-parts of an object.

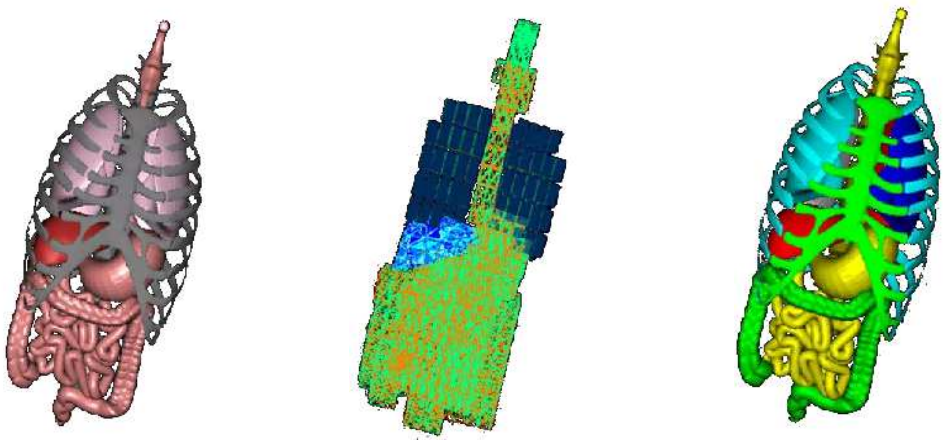
During this test a simple domain decomposition was enough to exploit the parallelism. More sophisticated partitioning approach could be used to decompose more complex objects. For instance, it could be a preprocessing step that generates a decomposed scene to be loaded by our simulator.

5.7.4 Medical Simulation

The SOFA Framework is targeted at medical simulations. We tested our parallelization approach on the simulation of a torso with most of its organs (Figure 5.15). Different simulation models are used. The bones were simulated as fixed rigid bodies. The lungs use hexahedral finite elements. The liver is simulated using tetrahedral finite elements. The intestine uses a mass-spring model.

The organs are subject to the gravity and to external forces from the user interaction. The organs are colliding all the time as all their movements are constrained in the space delimited by the rib cage. The time to compute each organ varies accordingly to the simulation model and the mesh resolution. For instance this scene is focused on liver surgery, so the most accurate model is used on the liver while a coarser mesh is used for the lungs and intestine.

We ran this test on a dual Quad Intel Xeon architecture, with a total of 8 cores. The sequential integration time is 50ms, and the parallel time using 8 cores is 14ms, resulting on a speedup of about 3.5. Note that we took this scene as it was and applied our object-level parallization scheme. It means that



(a) Visual Model (b) Behavior Model (c) Processor Distribution

Figure 5.15: Scene used in medical simulations tests.

this speedup is automatically available to any user, including non experts in parallelism.

A better performance could be obtained if we decompose the objects like shown in previous section. This scene has only seven objects that effectively counts in the final time, since the bones are rigid non-animated objects. It means that we have an eight-cores machine, but we only use seven of them during the simulation.

5.8 Summary

In this Chapter we presented an interface for mid-coarse grain parallelism that takes advantage of a static scheduling to generate independent partitions, and uses a dynamic scheduling to balance the load at run time. We extract tasks from the sequential algorithm to generate a data flow graph. This graph is then partitioned to be executed in parallel. When applied to a physical simulation, it allows to exploit the parallelism between different objects in a scene. The changes on the physical simulator are restricted to the system core, and all the physics routines are kept unchanged.

To parallelize complex applications, we introduced new control structures that represent loops in the graph. These loops can be partitioned and executed on multiple processors, giving access to extra coarse parallelism in the scene. Unlike traditional parallel loops, we can create loops whose number of iteration is dynamic. Additionally, conditional breaks are supported inside the loop body.

During the evaluation tests our approach exposed much more parallelism when compared to previous works. The data dependency analysis allows to avoid spurious synchronization imposed by strict recursive paradigms. With task partitioning we can bypass the need of a centralized controller that launches tasks. Partitions are automatically redeployed for the next iteration in a distributed way. This redeployment respects the affinity between a partition and a processor by using the same partition placement as the previous iteration.

This approach was tested on different scenarios going from similar independent objects to complex scenes. Tests show that we can obtain good performance results, achieving in some cases near optimal speedups. We outperformed previous works on a scene containing independent objects characterizing a massively parallel scenario. The data locality and decentralized task deployment allowed us to run two times faster than a pure dynamic scheduling approach.

A scene composed of objects with different sizes and simulated using different algorithms revealed the advantages of using dynamic scheduling for load balancing. The data dependency analysis exposes a maximum of parallelism at task level even when objects are colliding. By associating static and dynamic scheduling we obtained a speedup of about seven on a 16 core machine.

One of the weaknesses of coarse grain parallelism is the difficulty to adapt to scenarios where the number of parallel tasks is smaller than the number of resources. For this case we suggested a preprocessing step that provides a domain decomposition of the initial scene. By doing so we achieved a speedup of more than seven in a scene where a single soft tissue object was decomposed in 16 sub objects. This scene also reveals the advantages of data dependency analysis which enables to extract parallelism even when sub-parts of a decomposed object are tightly connected.

In all the scenes the gain on performance is obtained transparently to the physics developer. Each component can be internally implemented in a sequential way, and will continue to benefit from the parallel performance enhancements. However our parallel environment allows a given component to employ internally a fine grain parallelization, such as parallel loops on a multi core architecture, or GPU co-processors. In the next chapter we address the case of physical simulation on hybrid architectures composed by multiple processors and multiple GPUs.

Chapter 6

Hybrid Architectures

The works presented in this Chapter are part of an under reviewing paper.

Previous works presented in Section 4.2 are either only multi-CPU or consider multiple CPUs and a single GPU. Extending the work presented in the previous chapter, we propose a parallelization approach that takes advantage of the multiple CPU cores and GPUs available on SMP machines. The collision detection being performed efficiently on a single GPU [48], we focus here on the time integration step.

The work load of time integration varies in accordance with collisions: new collisions require the time integration step to compute and apply the associated new repulsion forces. As before, a first traverse enables to extract a data dependency graph between tasks. It defines the control flow graph of the application, which identifies the first level of parallelism: the parallelism between different bodies. The increasing usage of GPUs for physics simulation [56, 120, 47] makes possible to several tasks to have a CPU implementation as well as a GPU one using CUDA [101]. This GPU code provides a second fine-grain parallelization level to internally parallelize the computation of a body.

At runtime the tasks are scheduled following a two level scheduling strategy. At initialization and every time the task graph changes (addition or removal of collisions) the task graph is partitioned with a traditional graph partitioner and partitions are distributed to PUs (GPUs and CPUs are called Processing Unit). Then, to correct the work unbalance that may appear as the simulation progresses, work stealing is used to move partitions between PUs.

Our approach differs from the classical work stealing algorithm [51] as our stealing strategy takes into account the temporal and spatial locality. Spatial locality relies on the classical Owner Compute Rule where tasks using the same data tend to be scheduled on the same PU. This locality criteria is guaranteed during the tasks graph partitioning, where tasks accessing the same data are gathered in the same affinity group. Temporal locality occurs by reusing the task mapping between consecutive iterations. Thus, when starting a new time integration step, tasks are first assigned to the PU they ran on at the previous iteration.

CPUs tend to be more efficient than GPUs for small data-parallel tasks and vice-versa. We thus associate weights to tasks, based on their execution time. This weight is used by the PUs to steal tasks better suited to their capacities. Thanks to this criteria, PU heterogeneity becomes a performance improvement factor rather than a limiting one.

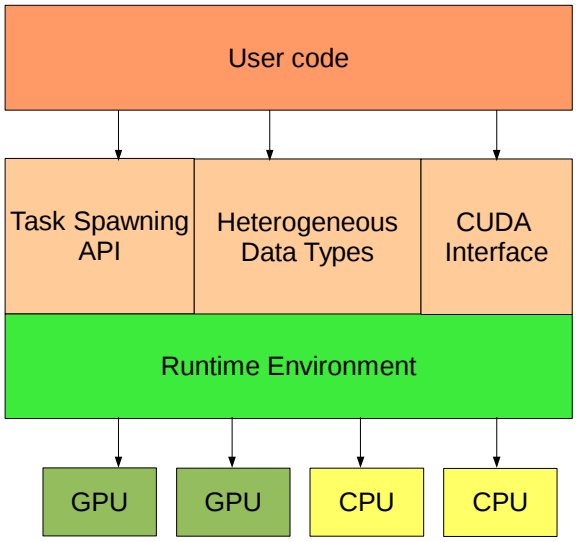


Figure 6.1: Organization of the multi-GPU abstraction layer on top of the run-time environment.

6.1 Multi-GPU Abstraction Layer

As seen in the previous chapter, to satisfy the requirements of collaborative developers, parallelism should be non-invasive i.e. the constructs used to express parallelism should have a reduced impact on the code. Additionally it should also be externalized, i.e., the scheduling and mapping of tasks on processors should not be embedded in the physical simulation code.

To attain these objectives on a multi-GPU context we rely on an abstraction layer that hides most of the parallelism complexity, delegating to the parallel environment the responsibility of orchestrating the different components execution. The parallel environment will then decide when and where to launch tasks accordingly to locality and load balancing criteria.

In Fig. 6.1 we introduce the abstraction layer we developed to ease deploying codes on multiple GPUs: multi-architecture data, to hide the data transfer between different devices; transparent multi-GPU launch, to run single-GPU components in multiple GPUs without changing the code; and multi-implementation tasks, to support spawning tasks having both a GPU and a CPU implementation.

6.1.1 Multi-architecture Data Types

The multi-GPU implementation for standard data types intends to hide all the complexity of data transfers and coherency management among multiple GPUs and CPUs. On shared memory multiprocessors all CPUs share the same address space and data coherency is hardware managed. In opposite, even when embedded in a single board, GPUs have their own local address space. To ease the utilization of such architecture, we developed a DSM-like (Distributed Shared Memory) mechanism that releases the

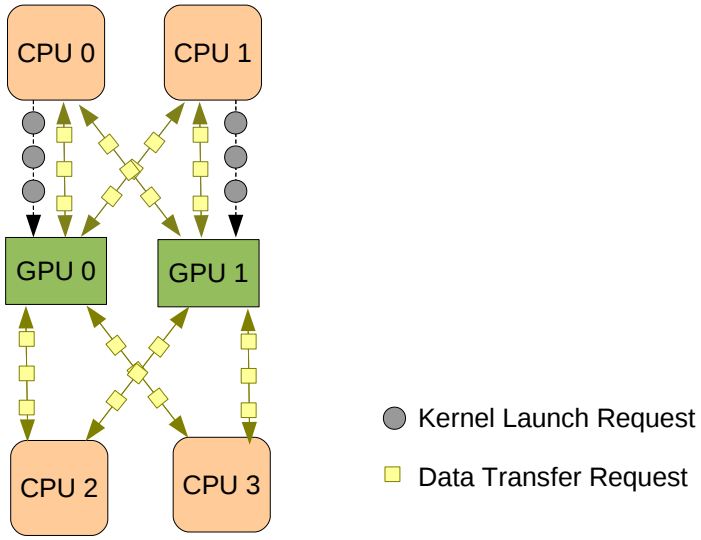


Figure 6.2: Data access on multi-GPU architectures. Only one CPU is attached to a GPU to launch kernels, but any CPU can access a GPU to read or write data.

programmer from the burden of moving data between a CPU and a GPU or between two GPUs.

When reading a variable, our data structure first queries the runtime environment to identify the processing unit trying to read the data. A bitmap is maintained on CPU-side to test if the PU has a valid data version. If so, it returns a memory reference that is valid in the address space of the PU requesting data access. If the local version is not valid, a copy from a valid version is required. For instance it happens when a processing unit accesses a variable for the first time, or when another processing unit has changed the data. This detection is based on dirty bits to flag valid version of the data on each PU. These bits are easily maintained by setting the valid flag of a PU each time the data is copied to it, and resetting all the flags but that of the current PU when the data is modified.

Since direct copies between GPU memories are not supported at CUDA level, data first have to transit though a CPU memory. Our layer transparently takes care of such transfers, but these transfers are clearly expensive and must be avoided as much as possible.

Recent GPUs also support accessing the CPU memory in a zero-copy way using DMA. It has the advantage of avoiding explicit memory copies, making possible to the GPU to directly change data on the CPU memory. However every access to the memory is translated into an access to the CPU memory, which is slower than accessing the local GPU memory.

Our multi-GPU data types also support the zero-copy mode. In this case a single memory pointer is managed. When a PU access a data, our data structure waits the end of the previous running kernel to ensure that the data in the zero-copy memory zone is valid. However the zero-copy mode must

```
struct TaskName  : Task::Signature<Shared_r<double>, Shared_w<int> > {};

template<>
struct RunCPU<TaskName> {
  void operator()( Shared_r<double> a,
                   Shared_w<int> b)
  {
      /* Implementation ... */
  }
};

template<>
struct RunGPU<TaskName> {
  void operator()( Shared_r<double> a,
                   Shared_w<int> b)
  {
      /* Implementation ... */
  }
};

int main(int, char*[]){
        Shared<double> inputData;
        Shared<int>    outputData;

        Spawn<TaskName>()(inputData, outputData);

}
```

Figure 6.3: Multi-implementation task definition. Top: Task Signature. Left: CPU Implementation. Right: GPU Implementation.

be avoided when the GPU access multiple times the same data. In these cases it is preferable to use the explicit memory copy mode, since data is transferred to the CPU only when necessary, instead of updating the CPU memory each times the GPU changes it.

6.1.2 Transparent Multi-GPU Kernel Launching

In a traditional multi-GPU scenario, the target GPU where a kernel will be executed is explicit in the code launching that kernel. This is constraining in our context as our scheduler needs to reallocate a kernel to a GPU different from the one it was supposed to run on, without having to modify the code. To hide this choice from the application we reimplemented part of the CUDA Runtime API. The code is written using the CUDA Runtime syntax to launch kernel, it is compiled and linked as usually done in a single GPU scenario. At execution time our implementation of the CUDA API is loaded and intercepts the calls to the standard CUDA API.

If the original code used to launch a kernel in a single GPU is thread-safe, the user can execute the independent CUDA kernels in parallel on different GPUs without changing the original warping code. When a CUDA kernel is launched, our library queries the runtime environment to know the target GPU. Then the execution context is retargeted to the right GPU and the kernel is launched. Once the kernel is finished, the execution context is released, so that other threads can use it, for instance when reading the results from a kernel execution. In Figure 6.2 we show how CPUs communicate with GPUs. Each GPU has a single controller CPU that is in charge of launching kernels into the GPU. On the other side memory reads and writes are done from any CPU requiring a data.

6.1.3 Architecture Specific Task Implementations

One of the purposes of our framework is to seamlessly execute a task on a CPU or a GPU. This requires an interface to hide the task implementation that is very different if it targets a CPU or a GPU. We provide a high level interface, as an extension of Athpascan for architecture specific task implementations (Fig. 6.3). First a task is associated with a signature that must be respected by all implementations. This signature includes the task parameters and their access mode: `Shared_r` for read-only data, `Shared_w` for write only data, `Shared_rw` for read and write data. This information will be further used to compute the data dependencies between tasks.

Each task implementation is coded as a functor object, i.e. an object that can be called like it was an ordinary function, using the () syntax. Like Athapascan, shared variables are passed as parameter to a `Fork` to spawn new task. The difference of the multi-architecture spawning is on the task identification. In the original Athapascan the Fork template is specialized to the functor that implements the task. In the multi-architecture context, the Fork template is specialized to the signature of the task.

There is thus a clear separation between a task definition and its various architecture specific implementations. The implementation for all the architectures must follow the same signature, so that any implementation can be used by the parallel environment without the need of additional wrapping.

Note that we expect that at least one implementation to be provided. If an implementation is missing, the task scheduler will simply reduce the range of possible target architectures to the supported subset. For instance, a CPU version is available in the large majority of the tasks, and it is used in the lack of a GPU version.

6.2 Scheduling on Multi-GPUs

The combination of static and dynamic scheduling employed for multi-core parallelism is also used on multi-GPU multi-CPU architectures. In this approach we first rely on a task partitioning that is executed every time the task graph changes, i.e. if new collisions or user interactions appear or disappear. Between two partitionings, work stealing is used to reduce the load imbalance that may result from work load variations due to the dynamic behavior of the simulation.

6.2.1 Partitioning and Task Mapping

On multi-GPU, the memory transfer costs more than on multi-CPU architectures. To enforce the spatial locality and minimize the memory transfers we decomposed the partitioning in two phases. First, the task graph is simply partitioned by creating one partition per physical object. Interaction tasks, i.e. tasks that access two objects, are mapped to one of these objects' partition. Each time the task graph changes due to addition or removal of interactions between objects (new collision or new user interactions), the partitioning is recomputed. As this partitioning is executed at runtime it is important to keep its cost as reduced as possible.

The granularity of this task graph is too fine, leading to large graphs costly to schedule. We thus rely on a higher level graph, the interaction graph, extracted from the physics simulator. Vertices represent

the simulated bodies and edges contacts between them. Using this interaction graph with METIS or SCOTCH, we compute a mapping of each partition that tries to minimize communications between PUs. The interaction graph is much smaller than the task graph, so it can be quickly processed.

A partition is then represented by a K-Thread 4.3.1. Tasks accessing the same simulated body are then inserted in the same K-Thread. Associating all tasks that share the same physical object into the same partition allows to increase affinity between these tasks. This significantly reduces memory transfers and improves performance especially on GPUs where these transfers are costly.

Physics simulation also shows a high level of temporal locality, i.e. the changes from one iteration to the next one are usually limited. Work stealing can move partitions to reduce load inbalance. These movements have a good chance to be relevant for the next iteration. Thus if no new partioning is required, each PU simply reuses the partitions executed during the previous iteration.

6.2.2 Dynamic Load Balancing

Like for multi-CPU simulation, we rely on a dynamic load balancing strategy based on work stealing to adapt to the dynamic changes on the scene, and on the processor load.

At the beginning of a new iteration each processing unit has a queue of partitions (an ordered list of tasks) to execute. The execution is then scheduled by the Kaapi [55] work stealing algorithm. During the execution, a PU first searches in its local queue for partition ready to execute. Here we say that a partition is ready if and only if the first task has all its read mode arguments already produced. If there is no ready partition in the local queue, the PU is considered idle and it tries to steal work from an other PU selected at random.

The original stealing algorithm presented in Section 4.3.3 has two stealing levels: K-Thread and task. In the hybrid scenario the steals only happen on K-Threads. The tasks in a same K-Thread being closely related we observed that stealing one task leaded to costly memory transfers.

Such work stealing provides an efficient load redistribution among idle and overloaded processors. However it does not ensure a good data locality when stealing tasks. The standard work stealing algorithm has being proved to have good locality on parallel recursive applications where tasks access data only from their parents [31]. This hierarchical data access pattern is absent for physics simulations where any simulated body can interact with any other in the scene. For instance, two colliding objects can remain simulated on different processors during a large number of iterations, which can lead to expensive memory transfers.

To improve performance for this kind of application we need to guide steals to favor gathering interacting objects on the same processing unit. One solution is to place all interacting bodies in the same processor at the deployment. This solution is optimal in terms of memory transfer as there is no data being accessed by different processors. However it can easily result in an unbalanced configuration if interacting groups are not uniforms.

Another solution called *locality guided work stealing* [6], assigns an affinity between tasks and processors, according to the data accessed. In addition to its local task list, a processor contains a second list of affinity tasks. When idle, the processor first tries to obtain some of these affinity tasks before attempting a classical steal.

Our approach combines both of them. The initial mapping, computed from the interaction graph, is already locality aware, i.e., after partitioning, the partitions are distributed over the processors in a way that minimize the interprocessor communication. To guide steals to favor gathering interacting objects on the same PU, we use an *affinity list* of PUs attached to each partition: a partition owned by a given PU has an other distant PU in its affinity list if and only if this distant PU holds at least one task that interacts with the partition. A PU steals a partition only if the PU is in the affinity list of the partition. We update the affinity list with respect to the PU that executes the tasks of the partition.

Unlike the *locality guided work stealing* in [6], this affinity control is only employed if the first task of the partition has already been executed. Before that, any processor can steal the partition. As we will see in the experiments, this combination of initial partitioning and locality guided work stealing significantly improves data locality and thus performance.

6.2.3 Harnessing Multiple GPUs and CPUs

Our target platforms have multiple CPUs and GPUs: the time to perform a task depends on the PUs, but also on the kind of task itself. Some of them may perform better on CPU, while others have shortest execution times on GPU. Usually GPUs are more efficient than CPUs on time-consuming tasks with high degree of data parallelism; and CPUs generally outperform GPUs on small size problems due to the cost of data transfer.

Following the idea of [28], we extended the work stealing policy to schedule time-consuming tasks on the fastest PUs, i.e. GPUs for large objects, and CPUs for small ones. Because tasks are grouped into partitions (Sec. 6.2.1), we apply this idea on partitions and not directly on tasks. Extending stealing to tasks would only increase memory transfers since tasks in a same partition are closely dependent.

The difficulty is to find a good metric to choose when a task should be executed on a CPU or on a GPU. In the simplest mode we tested, the user can statically set which simulated bodies have a CPU or GPU priority. This solution is acceptable when the number of simulated bodies is reduced, and requires a deep knowledge of the simulation behavior.

We also tried for instance to assign weight based on the associated body size. Though this approach proved efficiency if all bodies are simulated the same way, it fails for heterogeneous simulations mixing rigid and different soft models, where the computing cost depends not only on the object's size but also on the employed algorithm. The most interesting approach we tested is based on execution times. We take benefit of the iterative aspect of physics simulation and during the execution we collect the execution time of each partition on CPUs and GPUs. The first iterations are used as a warming phase to obtain the execution time on each architecture. Once we have both CPU and GPU times, the weight associated to each task is given by the $\frac{CPUTime}{GPUTime}$ ratio. Not having the best possible performance for these first iterations is acceptable for interactive simulations usually running several minutes. During execution the CPU and GPU times are constantly being updated taking into account the execution time from the previous iteration.

To choose the target architecture we implement a dynamic threshold algorithm. Partitions with $\frac{CPU\ Time}{GPU\ Time}$ ratio below the threshold are executed on a CPU, otherwise on a GPU. When a thief PU randomly selects a victim, it checks if its victim has a ready partition that satisfied the threshold criteria and steals it. Otherwise the PU chooses a new victim. To avoid PU starving for too long, the threshold

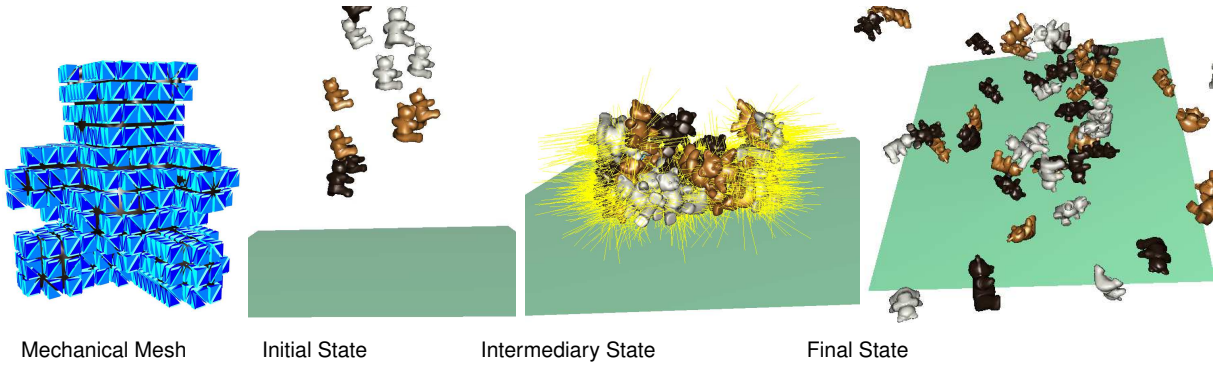


Figure 6.4: Simulation of 64 objects, falling and colliding under gravity. Each object is a deformable body simulated using Finite Element Method (FEM) with 3k particles.

is increased each time a CPU fails to steal, and decreases each time a GPU fails. This dynamic threshold allows a better parallel concurrent execution compared to maintaining a centralized queue of ready partitions sorted by their execution times.

The interaction graph partitioning and task mapping were not modified compared to the multi-GPU version. We did not investigate yet how partitioning could be improved to produce partitions adapted to each type of target processing unit.

6.3 Results

To validate our approach we used different simulation scenes including independent objects or colliding and attached objects. We tested it on a quad-core Intel Nehalem 3GHz with 4 Nvidia GeForce GTX 295 dual GPUs. Tests using 4 GPUs were performed on a dual quad-core Intel Nehalem 2.4 GHz with 2 Nvidia GeForce GTX 295 dual GPUs. The results presented are obtained from the mean value over 100 executions.

6.3.1 Colliding Objects on Multiple GPUs

Most of the works on task level parallel physics exploit the fact that different groups of colliding objects can be processed independently. However this approach can lead to load unbalance when independent collision groups have different amounts of bodies. For simulations such as in Fig. 6.4, where bodies are fully colliding, relying only on collision groups results on a weak speed up.

The first scene consists of 64 deformable objects falling under gravity (Fig. 6.5). This scene is homogeneous as all objects are composed by the same number of particles and simulated using a Finite Element Method with a conjugate gradient equation solver. At the beginning of the simulation all objects are separated, then the number of collisions increases reaching 60 pairs of colliding objects, before the objects start to separate again from each other under the action of repulsion forces. The reference average CPU sequential time is 3.8s per iteration. We remind that we focus on the time integration step. We just time this phase. Collision detection is executed in sequence with time integration on one GPU (0.04s per iteration on average for this scene).

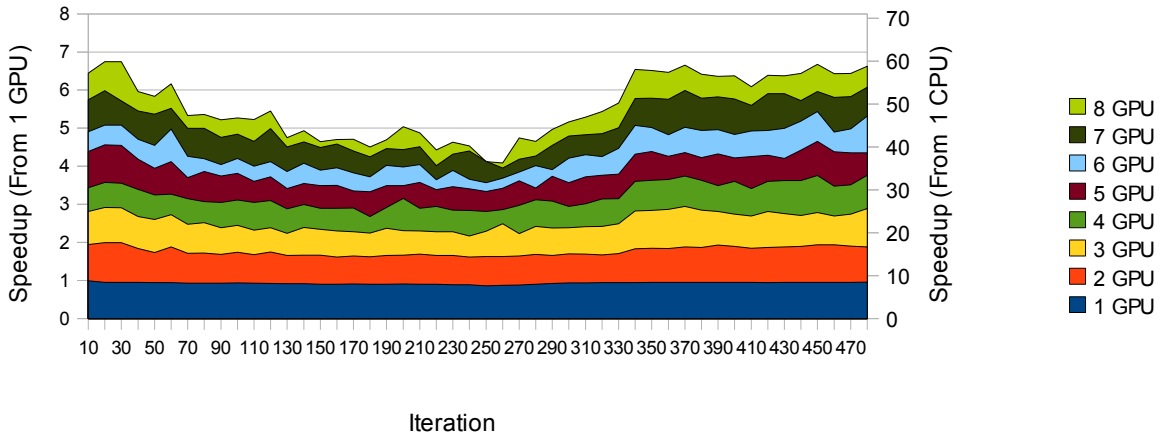


Figure 6.5: Speedup per iteration when simulating 64 deformable objects falling under gravity (Fig. 6.4) using up to 8 GPUs

We can observe that when objects are not colliding (beginning and end of the simulation) the speedup (relative to one GPU) is close to 7 with 8 GPUs. As expected the speedup decreases as the number of collisions increases, but we can still get at least a 50% efficiency (at iteration 260). During our experiments, we observed a high variance of the execution time at the iteration following the apparition of new collisions. This is due to the increasing number of steals needed to adapt the load from the partitioning. Steal overhead is important as they trigger GPU-CPU-GPU memory transfers.

The second scene tested (Figure 6.6) is very similar; we just changed the mechanical model of objects to get a scene composed of heterogeneous objects. Half of the 64 objects where simulated using a Finite Element Method, while the rest relied on a Mass-Springs model. The object sizes were also heterogeneous, ranging from 100 to 3k particles. Tests were done with and without collision response. We obtained an average speedup of 4.4, to be compared with 5.3 obtained for the homogeneous scene (Fig. 6.5). This lower speedup is due to the higher difficulty to find a well-balanced distribution due to scene heterogeneity.

6.3.2 Affinity Guided Work Stealing

We investigated the efficiency of our affinity guided work stealing. We simulated 30 soft blocks grouped in 12 bars (Fig. 6.7(a)). These bars are set horizontally and are attached to a wall. They blend under the action of gravity. The blocks attached in a single bar are interacting similarly to colliding objects.

We then compare the performance of this simulation while activating different scheduling strategies (Fig. 6.7(b)). The first scheduling strategy assigns blocks to 4 GPUs in a round-robin way. The result is a distribution that has a good load balance, but poor data locality, since blocks in a same bar are in different GPUs. The second strategy uses a static partitioning that groups the blocks in a same bar on the same GPU. This solution has a good data locality since no data is transferred between different GPUs, but the work load is not well balanced as the bars have different number of blocks. The third scheduling relies

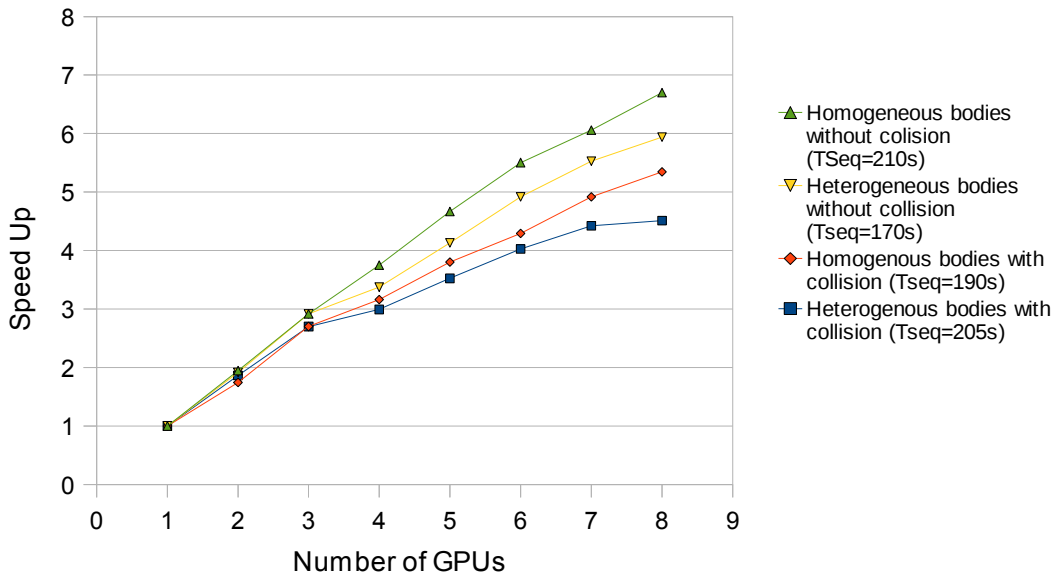


Figure 6.6: Heterogeneous scene results. Objects are simulated using either Finite Element or Mass-Spring model.

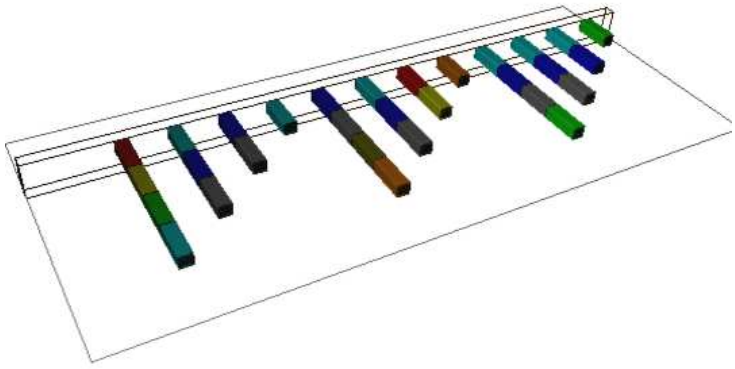
on a standard work stealing. It slightly outperforms the static partitioning for small blocks as it ensures a better load balancing. It also outperforms the round-robin scheduling because one GPU is slightly more loaded as it executes the OpenGL code for rendering the scene on a display. For larger objects, the cost of memory transfers during steals become more important, making work stealing less efficient than the two other scheduling. When relying on affinity, work stealing gives the best results for both block sizes. It enables to achieve a good load distribution while preserving data locality.

6.3.3 Involving CPUs

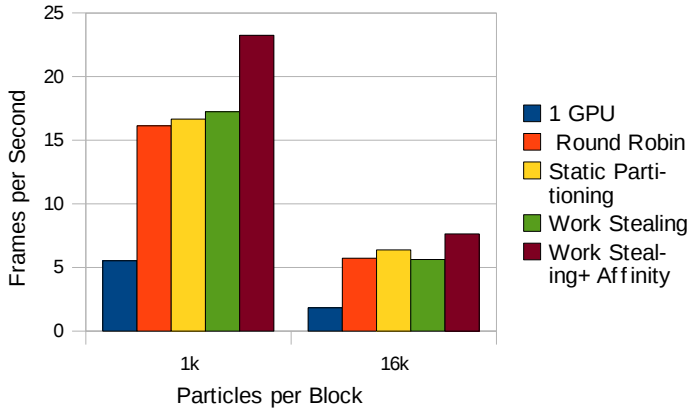
We tested a simulation combining multiple GPUs and CPUs in a machine with 4 GPUs and 8 Cores. Because GPUs are passive devices, a core is associated to each GPU to manage it. We thus have 4 cores left that could compute for the simulation. The scene consists of independent objects with 512 to 3 000 particles. We then compare standard work stealing with the priority guided work stealing (Sec. 6.2.3).

Results (Fig. 6.8) show that our priority guided work stealing always outperforms standard work stealing as soon as at least one CPU and one GPU are involved. We also get “cooperative speedups”. For instance the speedup with 4 GPUs and 4 CPUs (29), is larger than the sum of the 4 CPUs (3.5) and 4 GPUs (22) speedups. The reason comes from the SIMD GPU architecture. Processing a small object sometimes takes as long as a large one. With the priority guided work stealing, the CPUs will execute tasks that are not well suited to GPU. Then the GPU will only process larger tasks, resulting on larger performance gains than that if it had to take care of all smaller tasks.

In opposite, standard work stealing lead to “competitive speed-downs”. The simulation is slower with 4 GPUS and 4 CPUs than with only 4 GPUs. It can be explained by the fact that when a CPU takes a task that is not well-adapted to its architecture, it can become the critical path of the iteration, since tasks are not preemptive.



(a)



(b)

Figure 6.7: (a) A set of flexible bars attached to a wall (a color is associated to each block composing a bar). (b) Performances with blocks of different sizes, using different scheduling strategies

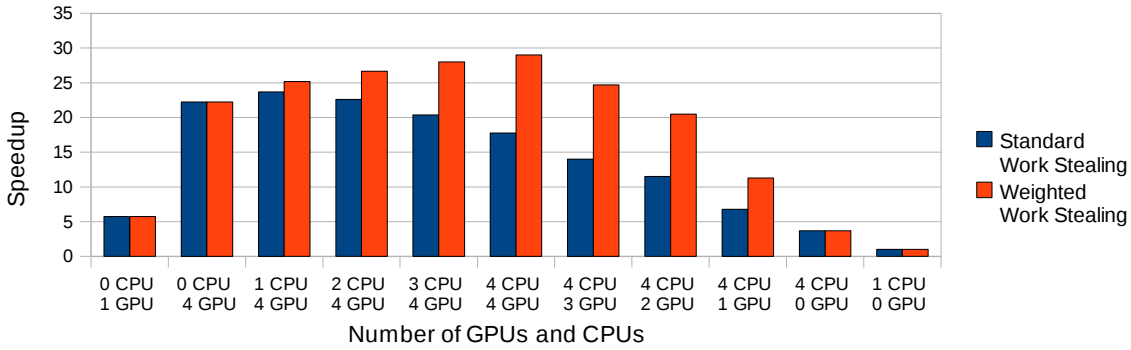


Figure 6.8: Simulation performances with various combinations of CPUs and GPUs.

6.3.4 Evaluating Scene Changes

As stated in Section 6.3.1 a high variance in performance is observed when the scene changes. Each variance peak coincides with a change in the task graph. To help understanding the origin of this variance, in Figure 6.10 we have the time spent on graph partitioning and memory transfers during a time step.

First, the task graph is repartitioned each time the scene changes. Second, the execution environment must adapt to the new scenario, resulting on memory transfers. It usually happens in the iteration just after the graph changes.

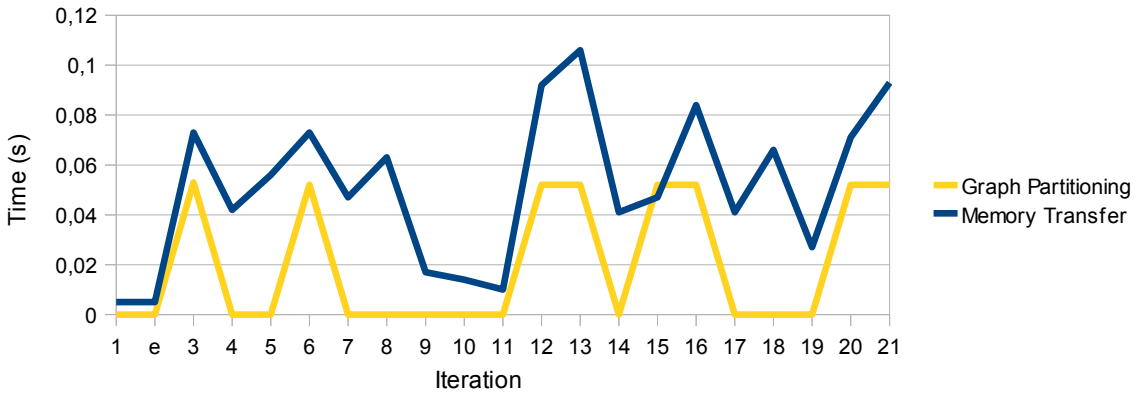


Figure 6.9: Decomposition of a time step.

On applications that support using an old graph while the new one is being partitioned, the graph partitioning cost can be amortized by computing the partitioning asynchronously. On the other side the memory transfer cost can hardly be reduced. This cost is strongly linked to the hardware architecture, and to the fact that data transfer between GPUs requires passing by the CPU memory.

The components that complete the execution of a time step are shown in Figure (6.10). This sample correspond to the iterations at the beginning of the scene used in Section 6.3.1 with eight GPUs. At this point of the simulation the number of colliding objects is low and the time integration still represents the largest portion of time.

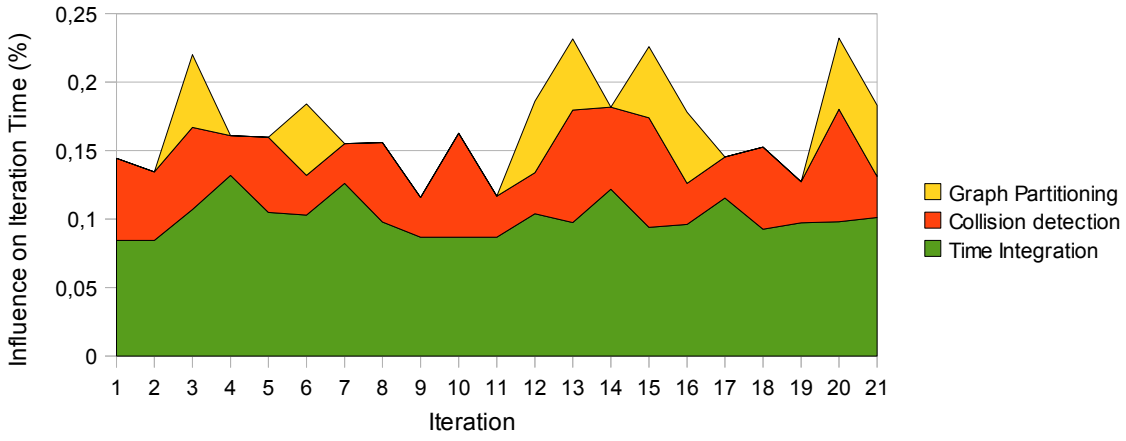


Figure 6.10: Decomposition of a time step.

6.4 Runtime Partition Generation

The implementation of the multi-CPU (Chapter 5 and multi-CPU/GPU simulation is based on previous works that considers static scheduling as a preprocessing step. The time to compute the partitioning is less critical than when recomputing the partitions during execution, like we do. Reusing these works allowed to validate new strategies on load balancing but the partitioning algorithm still requires some improvements.

Optimizations such as using an interaction graph (cf. Section 6.2.1) allowed to reduce the partitioning time, since this graph is smaller than the task graph. Also we can compile the graph as a background task, i.e. execute the previously computed partition while computing a new one. The partitioning cost can then be easily amortized. However this execution mode cannot be used in some applications since executing an outdated graph can affect stability.

We conceived a new partitioning algorithm that reduces the steps between the tasks creation and its execution. In the version used in these results, the task graph representing a whole iteration is created. Then it is evaluated to map task to processors, and create synchronization task (Section 5.4. Finally when partitions are generated they are deployed to the respective processors.

In the new approach we assume that all the tasks from a simulated body are placed in a same partition. It allows to the spawn function to directly deploy a task to the processor associated to the physics body. We rely on the result of the interaction graph partitioning to give an initial placement of the tasks. When a Fork is called, the task is pushed to the corresponding partition. Auxiliary data structures are used to detect at spawning time when additional tasks are needed to synchronize data between threads. Unlike previous version where a partition is deployed in a processor only after the partitioning phase, with the new approach the processor can then start executing tasks as soon as they are spawned. Preliminary tests on a prototype of this approach have shown that we can reduce the graph partitioning to one-fourth of the actual in a scene like the one in Figure 6.4.

6.5 Summary

In this chapter we extended the parallelization of physics simulations to multiple GPUs and CPUs. Due to the high cost of memory transfer between CPU and GPUs, we enforced the scheduling policy to tale into account the spatial and temporal locality. Temporal locality relies mainly on reusing the partition distribution between consecutive iterations, employing the same approach used on multi-CPU parallization. Spatial locality is enforced by partitioning the scene following the interactions between objects, and deploying colliding bodies in a same processor to minimizes memory transfers. We also guide steals toward partitions that need to access a physical object the thief already owns. For instance a processor will try to steal a body that is colliding with one that is already in the thief processor.

Moreover, in the heterogeneous context where both CPUs and GPUs are involved, we use a priority guided work stealing to favor the execution of low weight partitions on CPUs and large weight ones on GPUs. The weight can be computed based on the body size, or from the execution time obtained at runtime. The goal of the weighted stealing is to give each PU the partitions it executes the most efficiently. Experiments confirm the benefits of these strategies. In particular we get “cooperative speedups” when mixing CPUs and GPUs.

Chapter 7

Conclusion and Perspectives

7.1 Goals

During this thesis we studied the execution flow of physical simulation with the aim of improving interactivity. This kind of application involves an important collaborative work, with a set of algorithms from different natures interacting to produce a single simulated scene. Supporting collaborative works in a single application requires an important software engineering effort.

The Sofa Framework offers a well-defined interface for different physics components, allowing a seamless integration of new algorithms. Based on this framework, the aim of this thesis was to improve physics interactivity either by proposing new physics simulation algorithms or by using parallel architectures. Additionally the proposed solution must have a reduced impact in the code, so that it is accessible for collaborators that are not experts in parallelism

The well-defined interface of Sofa eases isolating the parallel environment from the physics code making possible to bring parallel architectures accessible to the physics simulation developers. The initial targeted architecture is shared memory multi-core machines. We also treated multiple specialized processors like GPUs, when a specialized version of an algorithm is available.

7.2 Methodology and Contributions

We started our works by evaluating the physics simulation algorithms to better identify the issues of this application class. As a result of this work, we noticed a lack of discrete time collision detection algorithm that are robust on large time-steps. This lack reduces the efficiency of scenes where objects are deeply interpenetrated from one step to the other. Our first contribution was to propose a **ray-traced collision detection**, that is able to separate interpenetrated objects.

The parallel works started with a prototype of a coarse grain parallelization of this collision detection algorithm using KAAPI, a parallel environment developed in the MOAIS team. In this approach the bounding boxes are computed in parallel. Using the bounding boxes the pair of objects are computed independently. The main difficulty comes from load imbalanaces due to the differences in size between

objects. Considering the physics pipeline, we observed that the most complex part to parallelize was the time integration step. Collision detection usually involves a single method that is used on all the objects. On the other side, during time integration, different algorithms are used to compute the internal state of each object, which can lead to more complex dependencies between tasks.

We developed a parallel version of the SOFA framework for multi-core architectures. We extended KAAPI to better adapt to the physics simulation execution flow. First we extract a task graph which is partitioned and deployed on the processors. One contribution of this work was **improving locality** by grouping tasks changing the same data in a same partition and redeploying a new iteration based on the data placement from de previous one. Another contribution were **dynamic loops** expressed in the task graph. It made possible to simulate more complex algorithm like conjugate gradient that requires a convergence loop. Also KAAPI has a load balancing strategy based on work-stealing, which was able to correct loads imbalanaces resulting from the different bodies size and algorithms.

Results of the multi-core parallelization show that we can obtain near optimal speedups in cases where objects are not colliding. We also observed an efficiency of about 50% when objects are colliding. Our extensions to KAAPI, allowed to better expose the parallelism of physics simulation, and the KAAPI data dependency analysis helped avoiding spurious synchronization between threads. All these improvements in performance were obtained with no changes in the physics algorithms, since we exploited the parallelism between different simulated bodies at a task level. New algorithms that are integrated in Sofa, and respect its framework interface can take advantage of parallel architecture without changing the code.

At the same time that multi-core architectures became popular, programmable graphics processing units emerged as an alternative architecture for data parallel algorithms. Physics simulation, such as deformable bodies, usually can be implemented using a data parallel approach. For instance, different particles of an object can be computed in parallel by applying the same operation on each one of them.

To take full advantage of machines composed by GPUs and CPUs, we extended our works on multi-core architectures to hybrid architectures composed by multi-CPUs and multi-GPUs. First we proposed an interface to specify **multi-architecture tasks** so that a scheduler can choose which implementation to execute: CPU or GPU. This interface extends the Athapascan interface that was used in the multi-core parallelization. Directly applying our works on multi-core architectures on hybrid architectures did not produce satisfying results. New constraints needed to be considered such as expensive memory transfer between CPUs and GPUs, and the choice of the task that better fits each architecture.

One contribution of this work on hybrid architectures was to **guide work-stealing** to better choose the victim tasks. First, to reduce the memory transfer between devices, we guide stealing to respect the scene layout, for instance by assigning an **affinity** between bodies that are colliding. An idle processor will preferably steal tasks that correspond to physics bodies that are interacting with bodies executed by the thief.

Second, we use a **priority guided** work stealing to choose the tasks that better fit to a processing unit. It favors the execution of low weight partitions on CPUs and large weight ones on GPUs. The weight can be computed based on the body size, or from the execution time obtained at runtime. The goal of the weighted stealing is to give each PU the partitions it executes the most efficiently.

Experiments show that we can get a speedup of about 4 when using 8 GPUs. Also we achieved to get cooperative speedups when combining CPUs and GPUs in a same scene, i.e. we obtained a speedup

that is higher than the sum of the individual speedup of each type of processing units. It was possible since the CPU computed the lighter bodies, leaving the heavier ones to GPU.

7.3 Perspective Works

The contribution of this thesis set the basis and validate many concepts for a parallel physics simulation using multi-CPU and multi-GPU machines. At the same time it opens new ways to investigate possible improvements. In this section we will detail some of the possible issues to better exploit parallel machines for physical simulation.

7.3.1 Locality-aware Collision Detection

Computing collision detection on GPU improved significantly its performance. It can be easily perceived when comparing a single CPU simulation with a single GPU simulation. However with our multi-GPU time-integration, using a single GPU for collision detection becomes a bottleneck. Adding more processing units only increases the time integration performance.

Using multiple GPUs for collision detection can alleviate this bottleneck. But parallelization alone cannot ensure a better performance. One weakness of our approach is to consider time integration and collision detection as completely separate computations. However, these two steps need to communicate at each time step, since collision detection needs to know the new object position computed by the time integration step. A multi-GPU approach for collision detection should minimize memory transfers between GPUs, because data locality is crucial to obtain good performance using multiple GPUs.

One way to minimize data transferring when going from time-integration step to collision detection is to use the data location at the end of the time integration step to determine where collision detection will be computed. The multi-GPU parallelization of the time-integration proposed during this thesis reduces memory transfers by guiding work-stealing to group colliding objects in a same processor. We can then compute the collision detection of the colliding bodies in the same processing unit that computed the time integration. To perform the broad-phase, we start by computing the bounding-box of each object locally, in a GPU that already contains its data. Then the bounding boxes are transferred to the CPU to compute the collision groups. Each collision group is then assigned to a GPU, accordingly to the bodies locality. Usually the number of new collisions from one time step to the other is low and most of the collision groups are maintained over iteration. It means that few objects will require memory transfer during collision detection.

7.3.2 Reduce Data Transfer in Interactions

As seen above, we can reduce the memory transfer between GPUs by preserving data locality when deciding where to run collision detection. However in some cases we cannot avoid transferring data between GPUs. For instance when a body is colliding with bodies whose time integration is done in different processing units. In this case two or more processing units requires the data from the same body to compute collision.

One way to reduce memory transfer overhead is by limiting the amount of data transferred. In a multi-model simulation we have different representations of the same object and these representations can have different levels of detail. For instance, we can have a mechanical model that is coarser than the collision model. The same can happen with the visual model. Sofa already has most of the basements for this kind of optimization (Section 2.5.1). Transferring only the coarser model and updating the finer one locally on each GPU can reduce the amount of data transferred specially when there is a huge difference in size between a models.

7.3.3 Dynamic Domain Decomposition

The grain of our parallel tasks are directly linked to the number of DOFs in a body. A task represents an operation on the entire body, and falling in a large object can affect the overall speedup. The origin of this drawback is that, at execution time, there is no mechanism to decompose the computation of a body.

Fine grain parallelism, like used for GPU parallelization, can accelerate the computation of a body. For instance we can reduce the computation grain by spawning new tasks from inside a huge one. This solution requires changing the internals of the physics algorithms. Another way to break down the computation is to decompose a body in sub-parts. It is the case of the mass-spring simulation in Section 5.7.3, where the different parts of a tissue are attached to ensure the conformity with the original body. A similar domain decomposition can be employed at runtime when the physical simulator detects that the existence of a huge body is affecting the performance.

7.3.4 Parallel Environment

Even if our target architecture is shared memory machines, the concepts of this work were implemented using a parallel environment that supports distributed memory (KAAPI). Consequently the specific contribution in parallelism of this thesis could eventually be applied to a distributed memory model. This support will require, naturally, an additional effort from the programmers to develop components that are compatible with a distributed memory architecture. Also the conditional loop management should be adapted to a distributed memory context to ensure the coherence between all the instances of a loop.

During our works on hybrid architectures we used external partitioners to obtain an initial work distribution that minimizes communication. The SCOTCH library is able to compute a partitioning by considering heterogeneous architectures, although we did not exploit this functionality, and processing units were considered as being homogeneous. On possible work would be to find metrics to describe an architecture composed of GPUs and CPUs and use SCOTCH to assign the right work to the right architecture based on the information obtained at runtime.

Another weakness of our multi-GPU implementations is to occupy one CPU for each GPU. It limits exploiting all the available power of the architecture, since a CPU consecrated to control a GPU is idle most of the time. This drawback can be circumvented by bringing the work-stealing control into the GPU so that it can obtain more work when idle without depending on a CPU. With the promising convergence among CPUs and GPUs, we also expect to have an architecture where memory transfers are not that costly as now, principally by integrating specialized processors in the mother board to access the main memory. This easier communication would enable to implement a more efficient work stealing between

CPU and GPU going towards considering the GPU as a standalone device and not a passive one.

Part III

Résumé en Français

Chapter 8

Introduction

La simulation physique interactive est une composante clé des environnements virtuels. Toutefois, le montant de calcul ainsi que la complexité du code augmente rapidement avec la variété, le nombre et la taille des objets simulés. Avec un code complexe, le développement de nouveaux algorithmes ainsi que le travail de collaboration deviennent difficiles. De plus, cette grosse demande de puissance de calcule peut compromettre interactivité de la simulation.

SOFA (Simulation Open Framework Architecture) cherche à réduire la complexité au niveau logiciel en fournissant une interface commune et bien définie pour les algorithmes physiques. Il permets une plus grande collaboration dans le domaine de de la recherche, grâce à la facilité de tester les algorithmes existante, et d'intégrer des nouveaux composants.

La complexité de la scène simulée est également réduite en combinant de composants simples pour générer des plus complexes. Bien que l'application cible principale de SOFA soit la simulation médicale, tel que la chirurgie virtuelle, des applications de différents domaines s'appuient elles aussi sur ce framework. Parmi les exemples nous pouvons citer les environnements virtuels immersifs, la génie des matériaux et la conception assistée par ordinateur.

Au cours de cette thèse nous avons étudié deux façons d'améliorer l'interactivité de la simulation physique. La première se base sur le développement de nouveaux algorithmes de simulation qui assurent une bonne perception de l'interactivité même si le taux de rafraîchissement est faible. La seconde consiste à tirer profit des architectures parallèles pour accélérer le calcul, ce que permets de simuler des scénarios plus complexes avec un temps réduit.

Une façon d'améliorer l'interactivité en modifiant l'algorithme de simulation est d'utiliser des grands pas de temps, ce que permets de rapprocher le temps virtuel du temps réel. Cette approche repose sur la façon dont l'utilisateur perçoit le résultat visuel. Même avec un faible taux de rafraîchissement l'utilisateur peut facilement interagir avec la scène si le pas de temps est assez grand pour garder le temps de simulation assez proche du temps réel. Cette approche peut être utilisé sur les méthodes d'intégration tels que Euler implicite avec un impact réduit sur la stabilité. Malheureusement, d'autres parties de la simulation physique sont plus sensibles. Par exemple, lorsque on fait de la détection de collision, des objets qui sont à séparés sur un pas de temps peuvent se traverser dans le pas de temps suivant. Lorsque cela se produit, l'algorithme de détection de collision doit être en mesure de séparer les objets interpénétrés pour être capable de profiter de grands pas de temps.

Certaines applications, telles que les jeux, peuvent dégrader la précision du calcul pour et être exécutés sur des ordinateurs de bureau moins puissants. D'autre part, la simulation scientifique nécessite généralement une puissance de calcul importante pour être interactive. Profiter des architectures parallèles, comme des machines multi-core, devient une nécessité. Toutefois, faire que cette puissance de calcul soit accessible à des non-experts en parallélisme est encore un défi. Paralléliser individuellement les algorithmes de simulation nécessite un effort important. Dans un travail collaboratif des nouveaux algorithmes sont constamment développées et, souvent, les développeurs ne sont pas experts en parallélisme. Pour offrir une alternative à ces développeurs nous proposons un approche qui exploite le parallélisme au niveau des composantes, ce qu'offre la possibilité aux développeurs d'écrire des composants de manière séquentielle. La difficulté est d'identifier les tâches indépendantes et au même temps faire en sorte que la charge de calcul soit bien distribué entre les ressources. Pour fournir une parallélisation transparent nous nous concentrons sur les machine à mémoire partagée. Ce que permets d'éviter la complexité et le surcoût supplémentaire des machines à mémoire distribuée. La complexité des aspects du parallélisme est ensuite délégué au framework parallèle.

Outre que les machines à mémoire partagée avec un nombre toujours croissant de processeur, l'émergence de machines avec de nombreuse unités de calcul fortement couplées comme les cartes graphiques, permettent la simulation physique interactive des scènes d'une complexité qui n'a jamais été réalisé jusqu'à présent. Des centaines d'éléments peuvent être calculé simultanément en utilisant des opérations du type SIMD. Les machines actuelles capable de déléguer du calcul à des cartes graphiques offrent normalement une mélange entre unités de calcul génériques (CPU) et d'unités spécialisés spécialisés (GPU). Une particularité de ces architectures est que les unités de traitement ne sont pas seulement hétérogènes en termes de puissance de calcul, mais aussi sur le modèle de programmation. Les CPUs sont programmés en utilisant un modèle *Multi-Instruction Multiple Data* (MIMD), alors que les GPU sont programmées comme *Single Instruction Multiple Data* (SIMD). La communication avec les unités de traitement spécialisées est coûteuse et peut avoir une grand impact sur la performance. En fait, des unités spécialisées (GPU) sont habituellement présentées sous forme d'une carte indépendante connectés sur un bus PCI-Express. Le transfert de données en utilisant ce bus PCI peut avoir une bande passante dix fois plus petite que l'accès a la mémoire de la carte graphique[1]. La difficulté est alors de tirer parti de tout le parallélisme disponible en déléguant du travail a l'unité de calcul la plus adapté et de minimiser le coût des transferts de mémoire.

8.1 Contributions

Un pipeline de simulation physique peut être décomposer en trois étapes principales: la détection de collision, l'intégration du temps et de l'affichage. La détection de collision compare la géométrie des objets à fin de calculer si les objets sont en collision. l'intégration du temps met à jour la position et les vitesses des objets selon les lois de la physique. L'affichage sert à montrer le résultat à l'utilisateur, soit sous forme visuelle en utilisant un écran, ou alors en utilisant des dispositifs haptiques ou tout autre périphérique de sortie. Les contributions de cette thèse se concentre sur les phases de détection de collision et d'intégration du temps.

La première contribution de cette thèse est un algorithme de détection de collision qui est robuste lors de l'utilisation de grands pas de temps. La qualité de la détection de collision est beaucoup plus dépendante du pas de temps que l'intégration du temps. Quand on utilise des grands pas de temps, deux

[1] Basé sur des tests sur une GeForce GTX 280

objets qui sont éloignés les uns des autres à un moment donné, peuvent se retrouver interpenetrés au pas de temps suivant. La détection de collision en continu peut éviter un tel problème en interpolant la trajectoire des objets et en calculant quand deux objets se touchent. Toutefois, le calcul de la détection de collision continue est assez coûteux, et dans la plupart des cas il n'est pas assez rapide pour être appliqué dans une simulation interactive. C'est pourquoi la plupart des simulateurs interactifs se basent sur les méthodes discrètes. Malheureusement, les approches traditionnelles, comme les méthodes par proximité, sont incapables de résoudre les cas où les objets se traversent profondément.

En raison de ce manque de solutions qui sont efficaces et robuste avec des grands pas de temps, nous avons développé une nouvelle méthode de détection de collision. Notre solution en temps discret est basée sur le lancer de rayon pour détecter quand deux objets entrent en collision. Un rayon est lancé vers l'intérieur de l'objet depuis chaque sommet, en suivant la direction de la normale. Une collision est détectée si la première intersection appartient à un polygone de surface extérieure d'un autre objet. Une force de contact entre l'origine du rayon et le point trouvé dans le deuxième objet est alors créé pour séparer les objets. Cet algorithme a l'avantage de calculer au même temps la détection de collision et la réaction à la collision, puisque le même rayon utilisé pour détecter une collision est utilisé pour créer un contact. En outre, il est robuste à des grands pas de temps car des objets interpénétrés peuvent être séparés par les contacts créés.

Pour être efficace, notre algorithme doit être précédée d'une phase de filtrage, généralement appelé *broadphase*, qui rejette des collision qui sont facilement détectés comme inexistantes. Par exemple, une *broad phase* permet de comparer les boites englobantes et déterminer si deux objets peuvent potentiellement entrer en collision. Notre algorithme est ensuite placé dans la *narrow phase* qui traite des objets potentiellement collisionés et compare ses polygones pour calculer la détection de collision a un niveau plus fin.

Pour améliorer le temps de calcul de détection de collision nous avons exploité les architecture multi-core en développant une version parallèle de cet algorithme. Dans notre algorithme les paires d'objets en collision peuvent être calculées indépendamment les uns des autres. Nous profitons de ce parallélisme inhérent pour distribuer les paires d'objets sur les différents unités de calcul. Nous appliquons ensuite un algorithme de équilibrage de charge basé sur le vol de travail pour éviter d'avoir des processeurs inactifs. Des tests avec une machine Quad-core ont montré que cette version parallèle est deux fois plus rapide par rapport à la séquentiel. Même si cette version est destiné à notre algorithme par lancer de rayon, il peut être appliqué à n'importe quel autre algorithme de détection de collision qui contient une *narrow phase* que soit *thread safe*. Cette généricité repose sur une parallélisation à gros grain, étant donné qu'on ne considère que le parallélisme entre les différentes paires d'objets en collision.

L'approche utilisée pour paralléliser l'étape de détection de collision repose sur un ordonnancement purement dynamique. Il s'adapte bien à cette première étape, car l'équilibre de charge est difficile à prévoir et aussi car la plupart des taches sont indépendantes et n'ont pas besoin d'une évaluation approfondie pour extraire du parallélisme. Paralléliser les autres étapes du pipeline de simulation est plus difficile puisque les dépendances de données sont intriqués.

Une autre contribution de cette thèse est la parallélisation de l'étape d'intégration du temps en utilisant un parallélisme de taches. Nous nous appuyons sur une approche hybride qui combine l'ordonnancement statique et l'équilibrage de charge dynamique. Lorsque on utilise l'ordonnancement statique toutes les tâches nécessaires au calcul d'une itération ainsi que les dépendances entre elles sont connues avant l'exécution. Nous pouvons évaluer les dépendances de tâches afin de mieux répartir le travail d'une manière qui minimise la communication. L'équilibrage de charge dynamique à son tour permet de redis-

tribuer la charge entre les processeurs.

L'exécution d'une itération de l'intégration du temps sur des machines à mémoire partagée fonctionne de la manière suivante. Avant l'exécution nous instrument le code en marquant les tâches et les données partagées entre elles. Ce code instrumenté est utilisé pour générer un graphe de dépendance de tâches qui contient toutes les opérations nécessaires pour effectuer un pas de temps de la simulation. Ensuite, le graphe est partitionné en utilisant une algorithme d'ordonnancement statiques pour améliorer la localité des données et de minimiser la communication entre les *threads*. Les partitions sont distribuées sur les processeurs, et un algorithme de vol de travail est utilisé pour répartir la charge sur plusieurs processeurs.

Le choix d'un parallélisme de taches à gros grain permets d'avoir un faible impact sur les algorithmes de simulation physique car le parallélisme est principalement extrait en changeant la coordination de l'exécution des taches. En outre, l'équilibrage de charge par vol de travail permets que certaines taches soient optimisés en utilisant un parallélisme de grain fin, sans que ça gène la distribution de charge. Toutefois, le vol de travail traditionnel ne donne pas de résultats satisfaisants sur des architectures hybrides composées par CPU et GPU. Le coûts de transfert de mémoire et les différences dans le modèle de programmation n'étant pas pris en compte lors du choix de la tâche à voler. Notre dernière contribution consiste à développer de nouveaux critères de vol de travail qui prennent en compte les particularités de cette architecture hybride. Nos nouveaux critères prennent en compte la localité de données et l'efficacité de l'algorithme sur chaque architecture. Une interface de haut niveau est utilisé pour créer des tâches qui ont de multiples implémentations. Nous pouvons spécifier, pour une même tache, une implémentation CPU et une implémentation GPU. Le choix de l'implémentation qui sera exécuté est délégué à notre mécanisme de ordonnancement. Cette interface de création de taches permets une séparation claire entre le code des applications et l'environnement parallèle. Des expériences en utilisant 4 GPU et 4 processeurs montrent qu'il est possible de faire en sorte que les CPU et les GPU coopèrent pour obtenir de meilleurs accélérations que ce qui est obtenu en considérant individuellement chaque architecture.

Chapter 9

Détection de Collision par Lancer de Rayon pour les Objets Déformables

La détection des collisions e est l'une des principales tâches de calcul de l'animation physique. Bien que des formes spéciales telles que des sphères ou des cubes permettent l'utilisation de méthodes optimisées, le cas général des maillages triangulaires est beaucoup plus complexe. Dans le cas des objets rigides, les approches les plus efficaces reposent sur les champs de distance. Chaque point d'un objet est testé contre le champ de la distance de l'autre. Si le point est à l'intérieur de l'objet, le point le plus proche à la surface est trouvé et une contrainte entre ces points est créé. Le calcul du champs des distance est une tâche de calcul coûteuse qui est effectuée au moment de l'initialisation des objets rigides, et défini par rapport à un référentiel local.

Lorsque les objets sont déformables, le champ de la distance doit être recalculée à chaque pas de temps, ce qui peut s'avérer trop coûteux pour des applications en temps réel. La stratégie la plus populaire est ainsi de détecter les paires de primitives géométriques à proximité, et de mettre en place des contraintes pour les séparer. Dans cette approche, les points de contact sont ceux qui ont une distance en dessous d'un seuil arbitraire donné par l'algorithme de proximité. Cependant, l'intégration en temps discret peut faire que les objets se traversent d'un pas de temps a l'autre, et quand une primitive pénètre plus loin que le seuil de proximité, il ne peut plus être identifié en tant que point de contact. Nous avons dans ce cas des réponses inefficaces à la collision que, parfois, ne sont pas capables de séparer les objets, comme illustré dans la figure 9.1.

Les problème de croisement de surfaces en raison de l'intégration en temps discret peut être atténué en utilisant des stratégies basées sur la anticipation des collision. Cependant, ces méthodes sont complexes. Leur convergence n'est pas claire et peut demander l'utilisation des petits pas de temps, tandis que des grands pas de temps sont préférables pour des applications en temps réel. Par conséquent, ils ont été principalement appliqué à des simulations de tissu non-interactive.

Pour les objets volumiques, un montant raisonnable d'intersection peut être visuellement acceptable. Une méthode de modélisation de contact robuste pourrait donc nous permettent de réaliser des grands pas de temps. Dans ce chapitre nous proposons un algorithme de détection de collision qui est robuste a l'utilisation de grands pas de temps, et permets de séparer les objets même quand ils se traversent.

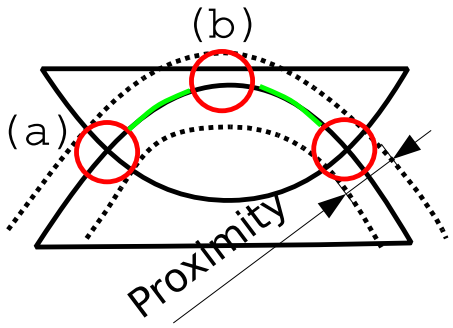


Figure 9.1: Problèmes avec la détection de collision basé sur la proximité quand les corps se croisent. (a):proximité à l'intérieur et l'extérieur du volume intersection s'annulent. (b): contacts indésirables peuvent être obtenus. Vert: une grande partie de la surface d'intersection est ignoré.

9.1 Detection

Si l'on considère un pipeline de collision, notre algorithme se place sur l'étape appelé *narrow phase*, car il fonctionne avec des paires d'objets qui sont potentiellement en collision. Elle nécessite une étape précédente, la *broad phase*, pour identifier des paires d'objets dont les boîtes englobantes sont en collision. En utilisant les pairs d'objets, notre algorithme trouve des pairs de points de collision (un point par objet). Les force de réaction sont ensuite appliquées à ce pair de points.

Pour identifier ces paires de points de collision, on choisit un sommet sur une surface de l'objet et on suit la direction inverse de la normale, jusqu'à trouver un point sur la surface de l'autre objet. Notre approche permet de résoudre les collisions, même si les objets si traversent et les triangles ne sont pas suffisamment proches pour être détectés avec une méthode par proximité. L'utilisation de la normale nous donne une bonne direction pour être utilisé avec des réponses par pénalité. Ainsi, le même algorithme nous permets d'obtenir la détection de collision et modéliser la réponse aux collision. Une fois que deux objets collationnent et des points de collision sont détectés, les forces utilisés pour répondre aux collision sont appliquées pour les séparer. Notre méthode ne dépend pas de données pré-calculées pour déterminer quels points sont en collision. Pour cette raison, il est bien adapté à des objets déformables, où les champs de distance sont trop coûteux pour être recalculée à chaque pas de temps.

Le chemin parcouru depuis un sommet d'un objet jusqu'à un point situé sur l'autre objet peut être représenté comme un rayon dont l'origine est sur le sommet et qui a une direction opposée à la normale du sommet. Pour accélérer la recherche d'éléments qui traversent ce rayon, nous avons stocké tous les triangles de chaque objets en collision dans un octree. La structure d'un octree nous permettent d'avoir un temps de recherche efficace indépendamment de la taille des triangles utilisés, ce qui n'est pas le cas pour une méthode qu'utilise des grille régulières.

Dans les sections suivantes, nous détaillons comment cette octree est construite, et comment nous utilisons cette structure pour trouver des paires de points de collision entre deux objets.

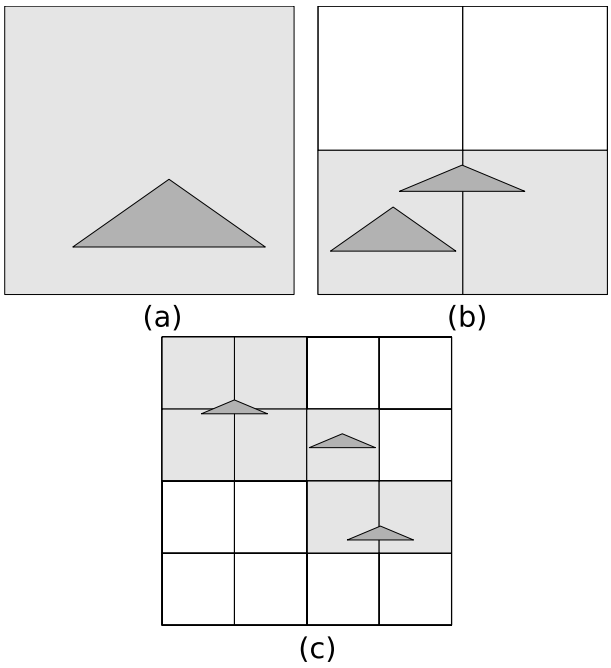


Figure 9.2: Version quadtree de l'algorithme de distribution de triangle. (A): un grand triangle placé au niveau de la première cellule, (b) 2 triangles stockés au deuxième niveau et (c) 3 triangles au troisième niveau.

9.1.1 Construction de l'Octree

Pour chaque objet potentiellement en collision nous construisons une octree contenant les triangles de la surface de l'objet concerné. Nous obtenons une structure de données spatiales qui nous permet de trouver facilement les triangles qui interceptent une région donnée. L'efficacité de l'octree dépend de la répartition spatiale des triangles. Une façon de construire l'octree est de fractionner une cellule si elle contient plus d'un triangle. Toutefois, cette approche ne permet pas de contrôler le nombre de cellules auxquelles un triangle appartient.

Notre algorithme garantit que chaque triangle est présent dans au maximum de huit nøeuds de l'octree. L'objectif est d'éviter de stocker un triangle dans un grand nombre de cellules. Notre méthode assure un bon équilibre entre la précision et le nombre de cellules à tester lors de la traversée de la octree. Pour atteindre cet objectif, un triangle est stocké au niveau le plus profond où la taille des cellules est supérieure à la plus grande dimension de la boîte englobante du triangle (voir figure 9.2). Certains triangles peuvent être stockées dans les nøeuds intermédiaires (non feuilles). C'est le cas des triangles de différentes tailles que se situent dans une même zone.

9.1.2 Lancer de Rayons

Nous lançons des rayons uniquement à partir des sommets situés dans la zone d'intersection des boîtes englobantes des deux objets, ce qui nous permet d'éliminer de nombreux tests. Cet algorithme se décompose en deux phases: recherche de paires de collision, et filtrage des résultats (voir l'Algorithme 4).

La phase de recherche consiste à prendre la direction opposé à la normal du point, et en suivant cette direction trouver un point situé sur la surface de l'autre objet. Les cellules octree sont visités en utilisant l'algorithme de parcours d'octree présenté par [111]. Chaque cellule de l'octree contient une liste de triangles qui interceptent cette cellule. Quand une cellule est visité, tous les triangles qu'elle contient sont testés contre les rayons en utilisant l'algorithme de [92]. Si un point d'intersection est trouvé, cet algorithme nous donne ses coordonnées et la distance entre l'origine du rayon.

Algorithm 4 Algorithme de detection de collision

Require: $Object1, Object2$
Ensure: pairs of colliding points between Object1 and Object2
  for each point1 in Object1 **do**
    $point2$=traceRay(point1, - point1.normal, Object2)
    if angle between point1.normal and point2.normal $\leq \pi/2$ **then**
      continue with the next point
    end if
    $point3$=traceRay(point1, - point1.normal, Object1)
    if distance(point1,point2) $\leq$ distance(point1,point3) **then**
      add collision pair to the collision response
    end if
  end for

Une fois une paire de points en collision trouvé, un sur chaque objet, nous vérifions la validité des contacts résultant comme illustré à la figure 9.3. La première vérification concerne l'angle entre les normales de deux points. Un angle aigu signifie que le rayon rentre dans le second objet au lieu de sortir. L'élimination des angles aigus permet d'éviter les erreurs de détection de paires de collision comme celle montrée à la figure 9.3(a), où deux rayons différents sont tracées à partir de O1, mais une seule est valable, vue que l'angle A est aigu. Appliquant des forces sur ces points ferait les objet se traverser encore plus.

Cependant, seulement l'élimination des paires de collision qui traversent le second objet de la face extérieure ne suffit pas. Nous devons nous assurer que le point que nous avons trouvé n'est pas en dehors de O1, car tout seul le critère de filtrage précédent peut générer des résultats ambigus. Un point qui fait partie de deux triangles peut avoir une normale qui répond au premier critère, même si le point est en dehors de l'objet entrant en collision. Figure 9.3(b) illustre la deuxième condition de validation d'un point en collision. Les rayons utilisés pour rechercher un point de collision sur O2 est réutilisé jusqu'à son intersection avec l'objet O1. Si le point trouvé sur O1 (point3) est plus proche de l'origine des rayons (point1) que le point sur O2, cette paire de collision est éliminé car le deuxième point est en dehors de l'objet 1. Les paires de collision qui satisfont tous les tests sont conservés pour être traités par la phase de réponse aux collision.

9.1.3 Auto-Collision

Les auto-collisions peuvent être détectées par une extension de notre méthode illustrée à la figure 9.4. Une auto-collision est détectée quand le rayon traverse deux fois la surface intérieure de l'objet. Ce test est plus cher parce que tous les sommets d'un objet sont testés.

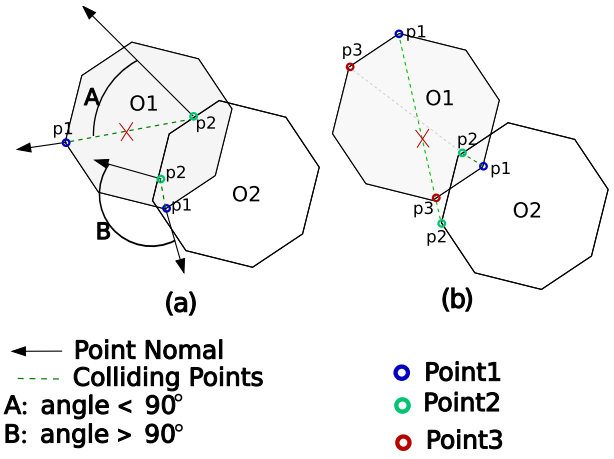


Figure 9.3: Validation des points de collision. Point1, point2 et point3 correspondent aux points identifiés dans l'Algorithme 4.

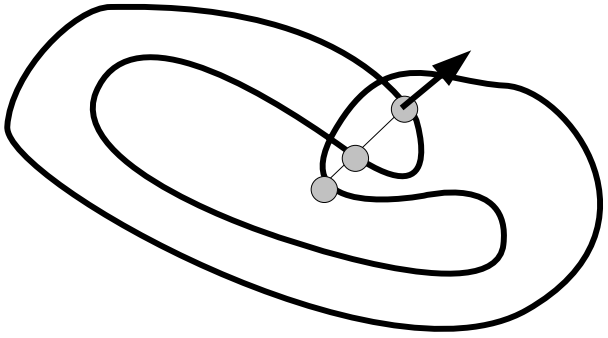


Figure 9.4: Détection des auto-collisions.

9.2 Réponse aux collision

Une fois que les collisions sont détectées et modélisées, nous appliquons une force de pénalité à chaque paire de points correspondants. La force est proportionnelle à la distance, et est parallèle à la ligne joignant ces deux points. Cela garantit que la troisième loi de Newton sur les forces opposées soit garantie. Nous appliquons la force directement au point de collision, et nous distribuons la force opposée sur les sommets du triangle associé à l'autre objet selon les coordonnées barycentriques du point d'intersection. Nous effectuons une intégration implicite du temps pour éviter les instabilités dues aux forces de contact normalement très fortes.

La direction de la force n'est pas nécessairement parallèle à la normale de l'objet qu'est entré en collision, comme l'illustre la figure 9.5, et quelques paires de contacts sont plus fiables que d'autres. En conséquence, les objets pointus peuvent subir à de forces tangentielles.

Nous multiplions l'intensité de la force par le cosinus de l'angle α de la figure 9.5(b). Cela réduit l'influence des forces de contact moins fiables.

Pour évaluer les forces de contact créé par notre algorithme, nous avons utilisé un cylindre traversé par un plan. La direction attendue pour la force résultante est une force verticale, comme le montre

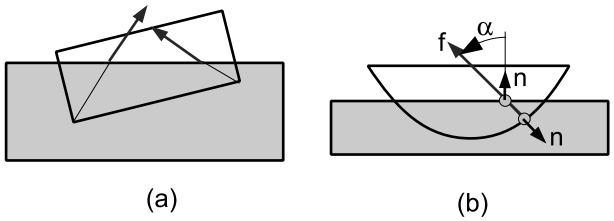


Figure 9.5: Force de contact résultante. Image (a), un objet pointu subit à un force nette tangentielle non-nulle. En (b), l'angle α est utilisé pour estimer la qualité du modèle de contact et le poids de sa force.

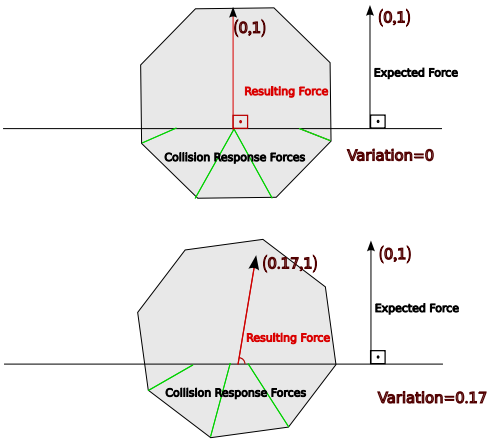


Figure 9.6: Un cylindre subit diverses forces tangentielles dues à la discretisation grossiere de l'objet

la figure 2D 9.6. En raison de la symétrie, les forces tangentielles devraient s'équilibrer et la force tangentielle devrait être nulle. Toutefois, la direction de la force résultante peut différer de la normale du plan, en raison de la discrétisation de la surface et de la position par rapport a la rotation du cylindre.

Pour mesurer la variation de la force résultante nous avons testé des cylindres avec des nombres de faces différents, allant de 10 à 150. Pour chaque cylindre nous avons testé 100 différentes positions de rotation, et pour chaque position nous avons mesuré les forces résultantes. Dans la figure 9.7, nous montrons la moyenne des variations d'un nombre variable de faces. Nous constatons que la qualité de la force résultante augmente avec le niveau de détail de l'objet. Pour un cylindre ayant seulement 10 parties, nous obtenons un écart type de 8% dans le pire des cas. Au fur et a mesure que nous augmentons le nombre de faces la variation se réduit à environ 1% . Cette variation dépend bien sûr de la forme de l'objet et les résultats on tendance a être meilleurs avec des objets lisses.

9.3 Résultats

Nous avons comparé l'efficacité de notre méthode avec une approche hiérarchique par proximité, analogue à celle proposée par [35]. Les algorithmes ont été implémentés en utilisant le framework de simulation SOFA [11]. La version parallèle de notre algorithme a été développé en utilisant KAAPI [55].

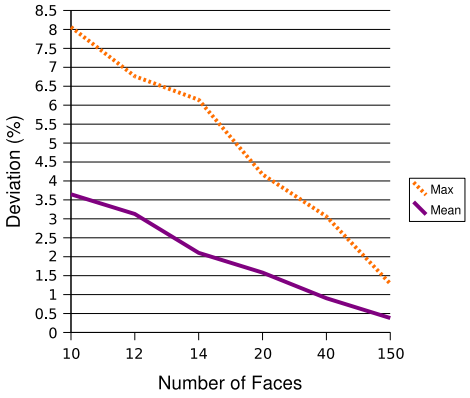


Figure 9.7: Ratio de la force tangentielle et normale, par rapport aunombre de faces du cylindre

.

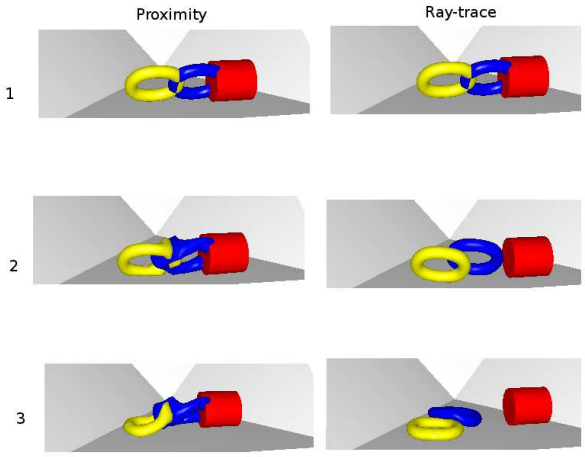


Figure 9.8: Une scène de test. Chaque tore est composé de 1600 triangles.

Le premier test consiste à une scène ou les objets se traversent depuis le départ. Nous observons la façon dont les algorithmes gèrent la séparation des objets. La Figure 9.8 montre la scène de départ suivi de la réaction produite par chaque algorithme. Notre algorithme de lancer des rayons parvient à séparer les objets, tandis que les objets ne sont que déformés et restent interpénétré dans le cas de l'approche par proximité. Tous les contactes crées par notre algorithme sont appliqués dans la bonne direction pour séparer les objets. L'algorithme basé sur la proximité se contente de pousser les triangles qui se trouvent dans une certaine limite de proximité. En conséquence, certaines pénalités sont orientées dans une direction opposée à celle qui devrait être utilisée pour séparer les objets.

Un autre avantage de notre approche est de permettre l'utilisation de pas de temps (dt) plus grands. Avec un *dt* grand les objets peuvent passer d'un état de non collision à un état ou ils se traversent profondément. Avec une approche basée sur la proximité, les objets traversés contiennent des triangles trop éloignées pour être détectée comme une collision. En utilisant la même scène sans intersection initiale (Figure 9.9), l'algorithme basé sur la proximité donne des résultats satisfaisants jusqu'à un *dt* maximale de 0,2 secondes, tandis que notre algorithme est capable de supporter un pas de temps de 0,45 secondes.

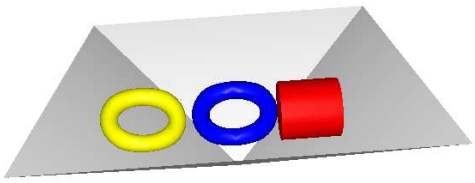


Figure 9.9: Scène utilisé pour la comparaison des performances.

Dans la scène illustrée à la figure 9.10, nous montrons une situation semblable où l'interpénétration des objets restreint les mouvements lors de l'utilisation d'une approche par proximité. Les anneaux commencent la simulation dans un état sans collisions, par contre des qu'ils collisionnent ils ne sont plus en mesure de glisser les uns sur les autres. Toutefois, avec notre méthode par lancer de rayon les anneaux peuvent se déplacer librement, même avec des interpénétrations. Il nous permet d'utiliser des pas de temps plus grands sans dégrader la qualité de la réponse aux collisions.

Lors de l'exécution des deux algorithmes sur une machine Xeon 2.5Ghz avec une scène comme celle de la figure 9.9, notre algorithme atteint 30 fps, alors que l'algorithme basé sur la proximité atteint seulement 12 fps. Ce résultat est principalement due à un nombre réduit de points de contact crées par le lancer de rayons. Des triangles proches qui ne collisionnent pas ne génèrent pas de contacts. Avec moins de contacts à appliquer le solveur converge plus rapidement.

En termes de scalabilité, notre algorithme se comporte comme prévu. La performance se dégrade de façon linéaire en fonction du nombre d'objets. L'élément fondamental de notre algorithme de détection de collision étant le triangle, la performance de l'algorithme dépend directement du nombre de triangles qui ont besoin d'être évalué. Dans la figure 9.11, nous présentons le temps nécessaire pour calculer 200 itérations avec un nombre variable d'objets dans la scène. Les objets utilisés sont des tores qui se collisionnent depuis le départ de la simulation.

Pour profiter des nouvelles architectures multi-core, nous avons élaboré une version parallèle de l'algorithme. Les paires d'objets en collision peuvent être calculé de façon indépendante. Nous profitons de ce parallélisme inhérent à notre algorithme pour distribuer les paires d'objets sur les différentes processeurs en utilisant une algorithme équilibrage de charge par vol de travail. Sur un processeur quad-core, cette algorithme est deux fois plus rapide que une exécution séquentielle. Le gain de performance est limitée par les parties du calcul qui restent séquentielles, comme la mise en place des forces de réponse aux collisions.

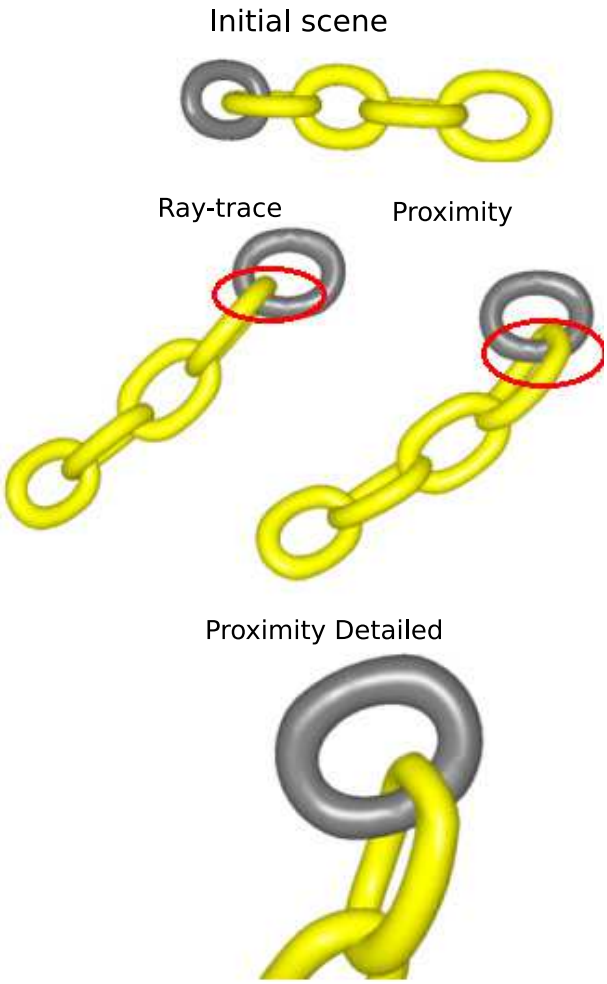


Figure 9.10: Test avec une chaîne déformable.

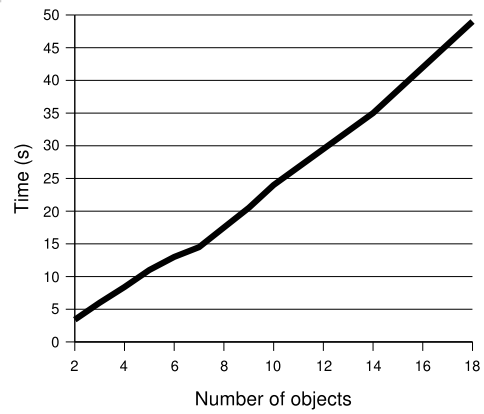


Figure 9.11: Évaluation de la performance avec un nombre variable de tores en collision, chacun d'entre eux est composé de 1600 triangles.

9.4 Conclusion

Dans ce chapitre nous avons présenté un nouvel algorithme de détection de collision pour les objets déformables. Cette approche est une alternative intéressante aux méthodes traditionnelles à base de proximités, notamment dans le cas des objets volumétriques déformable. Le temps de calcul sont plus courtes, et la robustesse nous permet d'appliquer des pas de temps plus grand. Le temps passé dans la construction d'un octree est compensée par l'accélération obtenue sur le lancer des rayons.

Chapter 10

Parallélisation de l'Intégration du Temps en Utilisant des Taches

Le pipeline de simulation physique est un processus itératif utilisé pour faire avancer la simulation en fonction du temps (Figure 10.1). Les variables sont normalement la position et la vitesse de chaque degré de liberté (DOF), stockées dans les vecteurs d'état. Le pipeline comprend une étape de détection de collision, utilisé pour créer ou supprimer des interactions entre les objets, basé. L'intégration du temps, qui consiste à calculer un nouvel état (position et vitesse) à partir de l'état initial et en intégrant les forces dans le temps. Enfin, le nouvel état de la scène est affiché

Dans nos travaux, nous nous concentrons sur l'intégration du temps. Dans cette phase, les DOF sont mis à jour par la résolution d'équations différentielles ordinaires. Notre approche combine la flexibilité et la performance, en utilisant une nouvelle approche pour paralléliser le couplage fort entre les objets indépendants. Nous nous basons sur le framework SOFA [12] où une scène simulée est découpé en groupes indépendants d'objets en interaction. Chaque groupe est composé des objets et de leurs forces d'interaction, suivi d'un solveur d'équations différentielles implicite. Chaque composent utilisé pour représenter un objet implémente des actions spécifiques liées aux forces, masses, les contraintes, les géométries et les autres paramètres de la simulation. .

Un pipeline de détection de collision ajoute et supprime des contacts en fonction des intersections. Basé sur les contactes il mets a jour les groupes d'objets, afin que chaque groupe puisse être traitée indépendamment des autres. Dans chaque groupe, le solveur d'équation traite une variété d'objets abstraits ainsi que des forces d'interaction, en traversant une structure de données en utilisant des visiteurs. Les visiteurs appliquent des tâches spécifiques à chaque composante, comme l'accumulation des forces sur les vecteurs d'état.

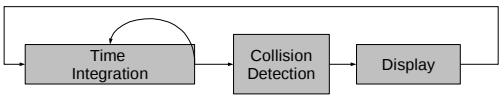


Figure 10.1: Simulation Pipeline

Algorithm 5 Simulation Pipeline with Explicit Time Integration

1: **loop**
2: Compute Force
3: Compute Acceleration
4: v+=a*dt
5: x+=v*dt
6: Collision Detection
7: Display
8: **end loop**

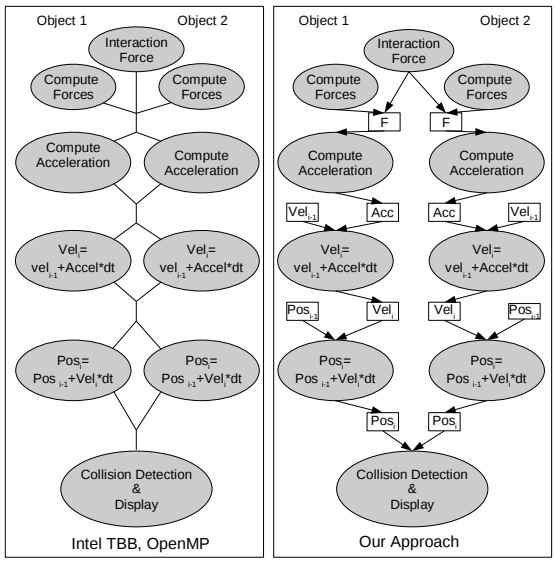


Figure 10.2: Le graphe de taches qui composent un pas de temps de l'integration explicite du temps 5. Gauche: Un approche recursif ajoute des synchronizations apres chaque operation. Droite:nottre aproche permets d'eviter les synchronization en utilisant une analyser de la dependece des donnees.

Pendant notre première tentative de parallélisation, chaque branche de la structure de données est traité en parallèle par les visiteurs déclenchées par l'algorithme de haut niveau. Malheureusement, en raison du caractère séquentiel des algorithmes de haut niveau (voir l'Algorithme 5), des synchronisations sont crées à la fin de chaque visite afin de s'assurer que les données utilisées par le visiteur suivant soient disponibles, comme illustré à gauche de la Figure 10.2, ce qui diminue considérablement le gain de performance. Pour contourner ce problème nous nous sommes appuyés sur une analyse du graphe de dépendance de données.

Une autre difficulté vient des boucles dynamiques, qui sont utilisés par les solveurs d'équation itératifs. Cette structure de contrôle n'était pas disponible dans notre environnement de programmation parallèle. Nous avons donc introduit des nouvelles instruction pour définir des boucle parallèle tel que présenté dans la Section 10.3.

10.1 Generation du Graph de tâches

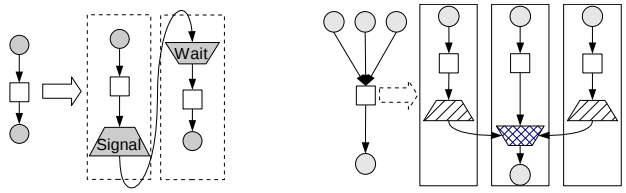
Le pipeline de simulation est organisé sous la forme d'une boucle qui fait avancer la simulation en fonction du temps (Figure 10.1). Nos travaux se concentrent sur la parallélisation de l'étape de l'intégration du temps. A son tour, cette étape peut elle aussi être composé de boucles internes, surtout lorsqu'elle s'appuie sur des méthodes comme l'intégration implicite du temps.

Pour extraire du parallélisme nous commençons par identifier les tâches et les dépendances entre elles. Les données accèdes par les tâches sont principalement des vecteurs représentant les états physiques d'un objet, comme les positions, vitesses et forces. Les tâches sont des opérations sur les vecteurs comme l'accumulation des forces, le calcul de positions de vitesses, etc. Les dépendances entre les tâches et les données sont représentées sous la forme d'arêtes d'un graphe. La Figure 10.2 correspond à un pas de temps d'intégration explicite lorsque deux objets entrent en collision. En réponse à la collision une force d'interaction est crée entre les deux objets.

Plutôt que exécuter directement les opérations, nous produisons d'abord un graphes de flux de données correspondant aux différentes opérations déclenchées par la résolution des équations. Chaque visiteur lancé par l'algorithme produit des tâches, et la séquence complète des opérations effectuées par l'algorithme est représentée comme un graphe qui modèle précisément la dépendance de données au niveau des composants. Avec une bonne répartition de ce graphe nous pouvons éviter les synchronisations indésirables presentées précédemment, et améliore considérablement les performances.

10.2 Partitionnement du Graphe

Une fois que nous avons un graphe de tâches représentant les opérations nécessaires pour la simulation, nous pouvons le partitionner ce graphe en assignant des tâches à un ensemble de processeurs. Dans la plupart des cas nous pouvons déduire le placement des tâche à partir de la structure de la scène. Dans la figure 10.2 toutes les tâches après le calcul des forces d'intégration modifient les données d'un seul objet. Dans ce cas, les tâches peuvent être automatiquement associés a une même partition. Les autres tâches qui ne sont pas directement associées à un objet, comme le calcul des force d'interaction, sont regroupés avec d'autres tâches qui accèdent un même groupe d'objets.



(a) Write Read Operation

(b) Cumulative Write

Figure 10.3: Tâches de contrôle employées pour garantir la cohérence d'accès aux données entre les différentes partitions. Avec un accès du type lecture/écriture le lecteur attends que les données soient produites par l'écrivain. Dans une mode d'écriture cumulative toutes les écritures sont effectuées dans un tampon temporaire qui seront par la suite accumulés par le réducteur.

Le regroupement des taches qui accèdent le même objet dans une même partition produit un partitionnement qui correspond à la structure de la scène. Il permet de rééquilibrer la répartition des objets sur les processeurs sans avoir besoin de partitionner le graphe. Nous pouvons déplacer un objet d'un processeur surchargé à un processeur inactive en réaffectant la partition lié à cet objet.

Après avoir partitionné le graphe de tâches, nous créons les *threads* qui seront exécutées en parallèle. Quand une tâche a besoins d'accéder des données produites par une autre partition on ajoute des tâches supplémentaires pour signaler la production des données. Dans un mode d'accès lecture/écriture comme montré sur la figure 10.3(a) une tâche *signal* est placée après la tâche écrivant la donnée. Cette tâche signale tous les lecteurs lorsque les données sont prêtes. Une tâche *wait* est placée juste avant le premier lecteur de chaque partition, pour attendre passivement le signal.

Pour supporter un écriture cumulative parallèle, qui se produit par exemple lors du calcul des forces (Figure 10.2), nous décomposons l'écriture des donnés en deux phases: l'accumulation et la réduction. Au début chaque écrivain accumule sa valeur à une zone de mémoire temporaire, ce qui permets une exécution parallèle (Figure 10.3(b)). Le *Reducer* attends tous les écrivains et somme les valeurs de tous les tampons temporaires.

Comme la plupart des étapes sont répétées d'une itération a l'autre, le graphe partitionné peut être réutilisé pour éviter les coûts de partitionnement du graphe et aussi renforcer la localité des données en limitant les transferts de données entre les itérations. Chaque fois que le graphe change il faut le partitionner à nouveau et réévaluer la dépendance de données pour garantir que l'exécution parallèle produit le même résultat que l'algorithme séquentiel.

10.3 Boucles Dynamiques

Comme indiqué sur a la section **??**, certains algorithmes d'intégration du temps doivent itérer jusqu'à satisfaire un critère de convergence. Pour être en mesure d'extraire du parallélisme de ce genre de boucle, il faut être capable de exprimer ces boucles dans le graphe.

Nous introduisons deux tâches spéciales, *EndLoop* et *BeginLoop*, utilisés pour délimiter la portée de la boucle. Toutes les tâches créées entre le *BeginLoop* et le *EndLoop* sont considérés comme partie du corps de la boucle et doivent être redéployées à chaque itération. Cette boucle peut être contrôlée par

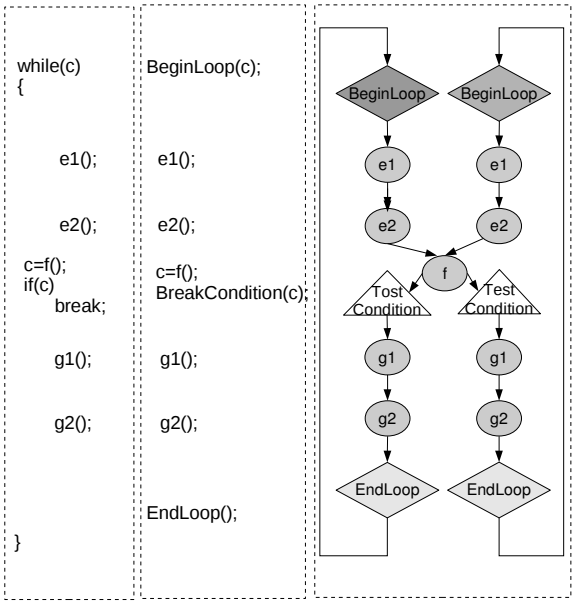


Figure 10.4: Boucle dynamique. Fonctions E1, E2, G1, G2 changent des objets multiples. A gauche: le code standard. Au milleu: notre code modifié. A droite: une représentation du graphe generé avec nottre approche.

une variable de condition ou d'un compteur d'itérations. La principale différence est que la variable de condition est partagée par toutes les boucles, tandis que un compteur d'itération peut être incrémenté localement par chaque boucle sans avoir besoin de synchronisation. Nous introduisons également une tâche *ConditionalBreak* qui permets de sortir de la boucle.

La condition est testée à chaque fois que les tâches *BeginLoop*, *EndLoop* ou *ConditionalBreak* sont exécutées. Si la condition de la boucle est toujours valable lors de l'exécution de la tâche *EndLoop*, toutes les tâches du corps de la boucle sont redéployés. Dans le cas contraire, nous devons sortir de la boucle. La prochaine tâche à exécuter est donc la tâche juste après le *EndLoop*.

Pour partitionner une boucle nous suivons le même système de placement utilisé pour les autres tâches, expliqué dans la section 10.2. Si une boucle est décomposé sur plusieurs partitions les tâches de contrôle de boucle sont répliquées sur tous les partitions, y compris le *ConditionalBreak*. Si la condition de rupture est une valeur booléenne, elle est traitée comme une donnée qui est partagée entre toutes les structures de contrôle de boucle. Sinon, dans le cas d'un compteur d'itérations, l'état peut être évalué localement par chaque partition, chaque partition peut s'exécuter librement, sans faire appel à des synchronisation supplémentaires.

Dans la figure 10.4, nous montrons une boucle exécutée sur deux partitions. La tâche *F* mets à jour la variable de condition à chaque itération, et toutes les boucles qui dépendent de cette variable sont des lecteurs de cette variable partagée. En traitant une variable de condition comme toutes les autres variables partagés, nous garantissons que le contrôle d'accès se fait automatiquement par le mécanisme *lécteur/évrivain* expliqué dans la section précédente.

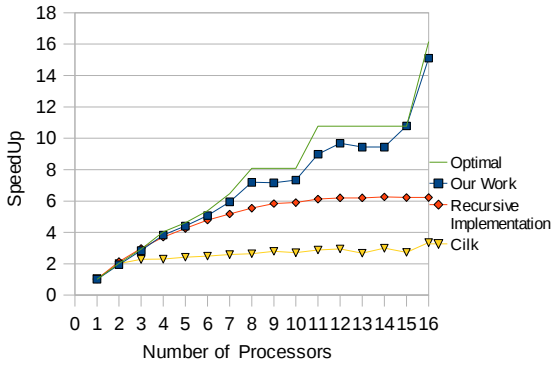


Figure 10.5: (
Speedup obtenu avec 64 bars identiques d'une taille de 16x4x4. T_1 = 150ms)

10.4 Résultats

Nous avons testé notre approche avec des différents scénarios de simulation; des objets indépendants et identiques qui et de scènes hétérogènes avec des collisions. Les tests ont été effectués avec une machine équipé de 16 cores (8 dual-core Opteron 2,2 GHz) avec 32 Go de RAM. Chaque processeur a un accès direct à 4 Go de mémoire et utilise le Hypertransport pour accéder à la mémoire des autres processeurs, ce qui caractériser un mode d'accès non uniforme à la mémoire.

Comme la parallélisation de l'étape de détection n'est pas le sujet principal de nos travaux, nous ne prenons en compte que les performance de l'étape d'intégration du temps. L'accélération est calculée en prenant le temps d'exécution séquentielle T_1 comme référence. Le temps partitionnement de graphe de travail (qui peuvent ne pas se produire à chaque itération) est toujours inclus dans le temps mesuré.

10.4.1 Objets Identiques Indépendants

Le premier test est une scène composée d'objets identiques simulés indépendamment, sans collision. Chaque objet est une barre souple avec une des extrémités attachées. L'objet se déforme à cause de la force de gravité. Ce test met en évidence la surcharge induite par la parallélisation.

Le test utilise 64 barres, chacune est composé de 256 particules qui sont simulés avec un méthode des éléments finis hexaèdriques (Figure 10.5). Cette taille d'objet est assez grand pour ne pas rentrer dans le cache, et en même temps suffisamment petit pour éviter d'avoir la mémoire comme un goulot d'étranglement.

Nous avons comparé notre approche avec une parallélisation en utilisant Cilk qui s'appuie sur le vol de travail, et une implémentation avec KAAPI en utilisant le parallélisme récursif avec un approche de diviser pour gagner comme celui de Cilk.

Dans les deux cas il n'a pas de boucle parallèle dynamique. Tous les algorithmes itératifs ont été exprimées à l'aide des boucles du langage de programmation, ce qui oblige à toutes les tâches de résynchroniser avant de commencer une nouvelle opération. Dans aucun des cas ils renforcent l'affinité entre les tâches et les processeurs. D'une étape à l'autre, il n'a aucune garantie que la tâche s'exécute sur le

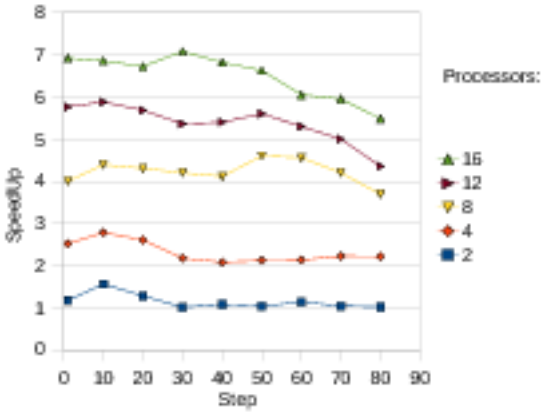


Figure 10.6: Speedup sur une scène hétérogène contenant des objets simulés en utilisant différents modèles mécaniques.

même processeur que l'itération précédente. Pour ce cas simple, l'approche récursif est plus lent que notre approche en fonction de la quantité limitée de parallélisme exposé qui n'est pas en mesure de compenser le surcoût de l'équilibrage de charge dynamique.

Notre implémentation est proche de l'optimal. Nous avons réussi à gérer les questions les plus importantes qui ne sont pas traités par les travaux précédents: les barrières de synchronisation et la localité de données d'une itération à l'autre.

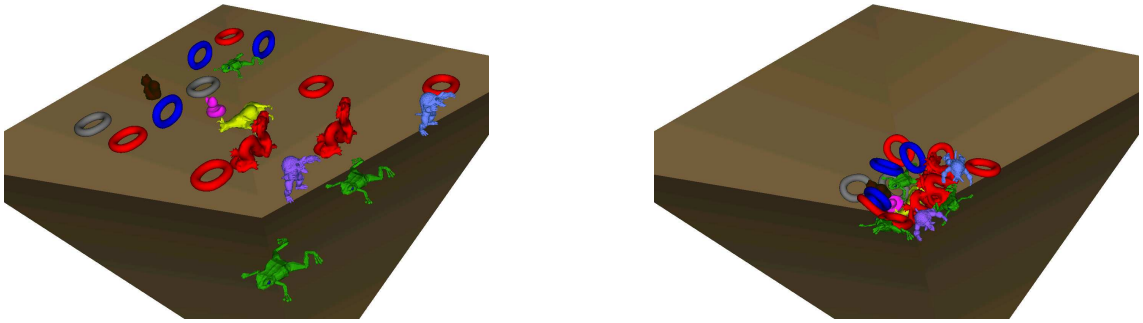
10.4.2 Scène Complexe

La parallélisation d'une scène composée d'objets indépendants peut être obtenue par des méthodes simples, comme la simulation de chaque objet dans un thread différent ou même dans un processus différent. Toutefois, lorsque les objets entrent en collision, une telle méthode ne peut profiter d'une quantité réduite de parallélisme. C'est pour ce genre de simulation que notre approche se montre efficace.

Dans la figure 10.7, nous avons l'état initial et final d'une scène d'objets hétérogènes qui subissent à la force de la gravité et à des collisions. Les différents objets ont des maillages surfaciques qui vont de 400 à 2.500 triangles. La méthode employée pour simuler l'objet varie: il y a des corps rigides et corps mous utilisant des masse-ressorts, éléments finis ou modèles de grilles déformables.

Durant les premières étapes il ya presque pas de collision entre les objets, et comme expliqué à la section **??**, ils peuvent être résolus par des instances distinctes du solveur. En plus, nous obtenons une accélération importante étant donné que peu de tâches accèdent à des données de d'objets différents. Cependant, alors que nous approchons de l'état final, les objets se rapprochent, et à la fin ils font tous partie d'un seul groupe de collision.

Cela signifie que tous doivent partager la même boucle itérative, car l'instance du solveur employés sur tous les objets est la même pour éviter les instabilités (*couplage fort*). En raison de la collision, des nombreuses tâches accèdent à plusieurs objets, créant des points de synchronisation pour partager les données.



(a) Initial State (b) Final State

Figure 10.7: Scene used for the complex simulation tests. Objects are simulated using different methods, like Finite Elements, Springs and Regular Grids

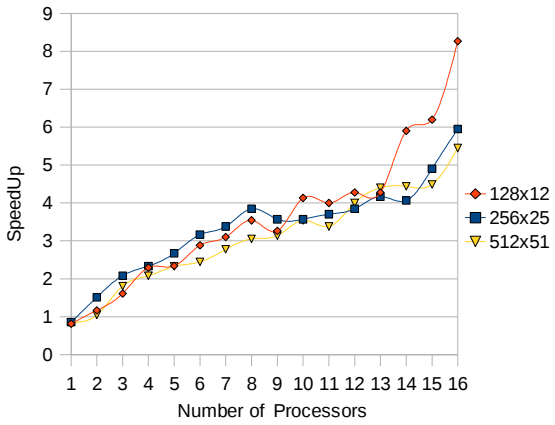


Figure 10.8: Speedup en utilisant une système de masse-ressort par décomposition de domaine.Le temps séquentiel pour chaque cas est: 125ms (128x128), 500 ms (258x258), 2.2s (512x512)

Lorsque les groupes de collision changent, le graphe de tâches est mis à jour. Le coût de cette procédure est proportionnelle au nombre de objets dans la scène. Dans la scène de la figure 10.7 Le coût moyen de mise à jour du graphique est proche de 30ms, ce qui représente environ le même temps requis pour exécuter un pas de temps. Ce surcoût est généralement pas perceptible. Il est compensé par l'amélioration de la performance globale et il est souvent amorti après quelques pas de temps.

L'accélération obtenue ici ne nécessite aucun effort de la part du développeur de l'application, ni du développeur de l'algorithme physique. Les algorithmes physiques peuvent évoluer indépendamment du code parallèle, ce que réduit le décalage entre le développement de l'algorithme séquentiel et l'exécution de cet algorithme dans une architecture parallèle.

10.4.3 Décomposition du domaine

Notre approche vise un parallélisme à gros grain, sans prendre en compte l'implémentation interne d'une méthode donnée. Dans une scène avec peu d'objets ou avec un objet qui demande un grand effort de calcul, il est plus difficile d'obtenir des gains performances importants. Comme SOFA permet d'avoir des objets fortement liés, on peut explicitement décomposer un objet de grande taille en objets plus petits. SOFA garanti que les deux simulation physique seront identiques. Des contraintes assurent que deux points appartenant à des objets différents seront considérés comme un seul point tout au long de la simulation. Pour évaluer ce type de décomposition de domaine, nous avons utilisé un système de masse-ressort qui est coupé en petits objets comme le montre la figure 10.9. Avec cette simple décomposition, l'accélération peut être significative (Figure 10.8).

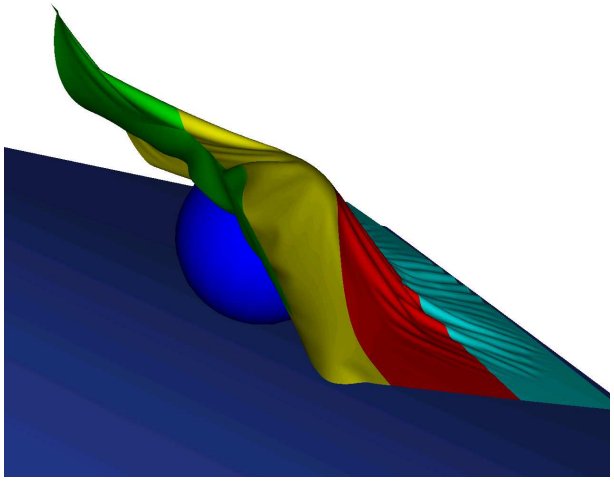


Figure 10.9: Décomposition de Domaine: pour exposer plus de parallélisme un tissu est subdivisé en bandes horizontales.

Nous avons obtenu de meilleurs résultats avec des objets plus petits en raison de l'surcoût introduit par les accès mémoire sur les objets les plus grands.

Notez que comme pour le test précédent, tous les objets partagent la même boucle itérative pour la phase de convergence du gradient conjugué.

10.4.4 Simulation Médicale

Le dernier test met l'accent sur une scène plus réaliste simulant un modèle d'un torse avec des organes. Les os ont été simulées comme des corps rigides, les poumons et le foie sont simulés par éléments finis, et les intestins avec des ressorts. Nous avons exécuté ce test sur une architecture dual Quad Intel Xeon, avec un total de 8 cores. Le temps d'intégration séquentiel est de 50 ms, et le temps parallèle avec 8 cores est de 14ms, ce qui entraîne une accélération d'environ $3,5$.

Notez que cette accélération est automatiquement disponible à tout utilisateur, y compris des non-experts en parallélisme.

10.5 Conclusion

Dans ce chapitre nous avons présenté un framework qui tire parti du parallélisme entre les différents objets dans une scène. Nous extrayons des tâches de l'algorithme séquentiel pour générer un graphe de flot de données. Ce graphe est alors partitionné pour être exécuté en parallèle. Les changements sur le simulateur physiques sont limités au noyau du système, et toutes les routines physique sont restées inchangés.

Pour pouvoir paralléliser des scènes plus complexes, nous avons introduit de nouvelles structures de contrôle pour représenter les boucles dans un graphe. Ces boucles peuvent être partitionnés et exécutées sur plusieurs processeurs. Contrairement aux boucles parallèles statiques, nous pouvons créer des boucles dont le nombre d'itérations est dynamique. En outre, des arrêts conditionnelles peuvent être crées à l'intérieur de la boucle.

Cette approche a été testée sur différents scénarios allant des objets identiques et indépendants à de scènes plus complexes. Les tests montrent que nous pouvons obtenir des bonnes résultats, ce que nous a permis d'obtenir, dans certains cas, une accélération presque optimale. Dans toutes les scènes, le gain de performance est obtenue de manière transparente au développeur des algorithmes physiques.

Chapter 11

Simulation Physique Intéractive sur une architecture Multi-CPU et Multi-GPU

11.1 Introduction

La simulations physique interactive est un des éléments clés des environnements virtuels. Cependant, la quantité de calculs ainsi que la complexité du code augmente rapidement avec la variété, le nombre et la taille des objets simulés. La difficulté est alors de tirer parti efficacement de ces architectures.

Nous présentons dans ce chapitre une approche de parallélisation qui tire parti des architectures multi-CPU et multi-GPU sur des machines SMP. Notre implémentation se base sur la bibliothèque open source SOFA conçu pour offrir un degré élevé de flexibilité et un calcul de haute performance.

Nous avons développé une parallélisation multi-GPU et multi-CPU pour l'intégration du temps. Tout d'abord une première exécution de l'algorithme de simulation nous permets d'extraire un graphe de dépendance des données. Il définit le graphe de flot de contrôle de l'application, qui identifie le premier niveau de parallélisme. Plusieurs tâches ont une implémentation CPU ainsi que un implémentation GPU en utilisant CUDA [101]. Ce code GPU offre un second niveau de parallélisme à grain fin. Pendant l'exécution, les tâches sont ordonnancées selon un approche à deux niveaux. À l'initialisation et à chaque fois que le graphe de tâches change (ajout ou la suppression des collisions), le graphe de tâches est partitionné avec un partitionneur de graphe et les partitions sont distribuées aux unités de calcul. Ensuite, pour corriger des éventuels deséquilibrages de charges qui peuvent apparaître pendant la simulation, un algorithme de vol de travail est utilisé pour déplacer les partitions entre les unités de calcul.

Notre approche diffère de l'approche classique de vol de travail [51] car il prends en compte la localité spatiale et temporelle des données. La localité spatiale repose sur l'approche appelé *Owner Compute Rule* qui consiste à exécuter sur une même unité de calcule les tâches qui utilisent le même ensemble de données. Ce critère de localité est garanti par le partitionnement du graphe de tâches, où les tâches accèdent aux mêmes données sont groupées dans une même groupe d'affinité. La localité temporelle os obtenue en conservant le placement des tâches entre deux itérations consécutives. Quand une nouvelle itération est entamé les tâches sont attribués à la unité de calcul ou elles ont été exécutés à la itération précédente.

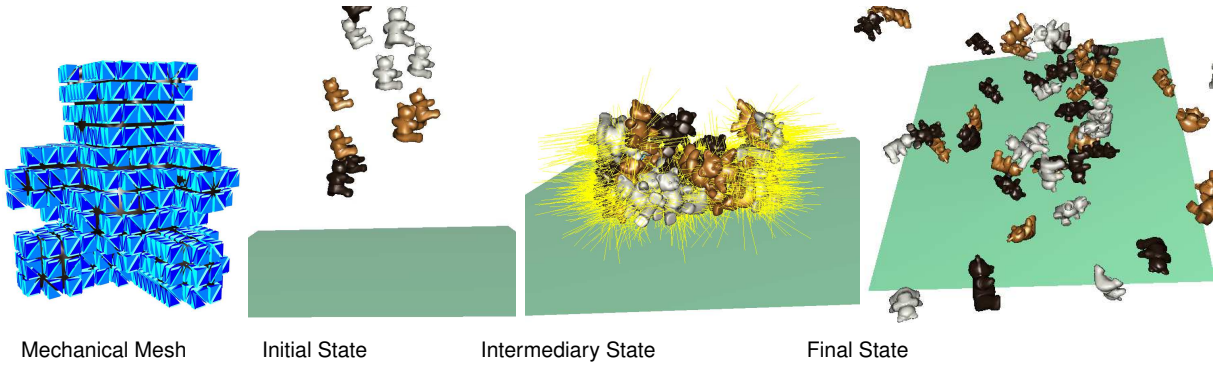


Figure 11.1: Simulation de 64 objets qui tombent avec des collisions. Chaque objet est un corps déformables simulés à l'aide d'un méthode des éléments finis (FEM) avec 3k particules.

Le processeurs ont tendance à être plus efficaces que les GPU pour des petites tâches et vice-versa. Nous associons ainsi des poids à des tâches en fonction de leur temps d'exécution. Ce temps est utilisé par les unités de calcul pour voler la tâches le mieux adaptées à leurs capacités.

Les tests montrent que en utilisant une scène complexe composé de 64 objets en collision, totalisant plus de 400K éléments FEM peut être simulé sur 8 GPU en 0.082s par itération au lieu de 3.82s sur un CPU. Une scène hétérogènes avec au même temps des objets complexes et simples peuvent exploiter efficacement toutes les ressources d'une machine avec 4 GPU et 8 CPUs pour calculer l'intégration du temps 29 fois plus rapidement qu'avec un seul processeur. L'ajout des 4 cores CPU augmente les performances de simulation de 30%, nettement plus que les 5% prévu, car les CPU déchargent les GPUs des tâches petites, ce qui augmente l'efficacité des GPU.

11.2 Simulation Physique

La simulations physique présente un défi pour le calcul de haute performance. Des calcul des natures différents avec des coût de calcul non prévisibles sont demandés. Dans certains cas pendant un pas de temps donné les tâches peuvent accèder à une grande quantité de données, avec un graphe contenant des milliers des tâches interdépendantes. Toutes ces opérations doivent être exécutées dans un intervalle d'environ 40 ms pour que l'application puisse rester interactive.

11.3 Couche d'abstraction Multi-GPU

Nous allons d'abord introduire la couche d'abstraction, qui vise faciliter le déploiement du codes sur plusieurs GPUs.

11.3.1 Types de données Multi-architecture

Le multiprocesseurs à mémoire partagée ont tous un même espace d'adressage et la cohérence des données est géré par le matériel. De son coté les GPUs, même quand intégrés dans une même carte, ont

```
struct TaskName : Task::Signature<cdnst double*, int * > {};

template<>
struct RunCPU<TaskName> {
  void operator()( const double * a, int *b)
  {/* Implementation ... */ }
};

template<>
struct RunGPU<TaskName> {
  void operator()( const double * a, int *b)
  {/* Implementation ... */ }
};
```

Figure 11.2: Définition d'une tâche multi-implementation. Haut: Signature de la tâche. A gauche: L'implémentation CPU. A droite: l'implémentation GPU.

chacun leur propre espace d'adressage. Nous avons développé un mécanisme pour gérer la mémoire distribué pour cacher de l'utilisateur toute la complexité des transferts de données et la gestion de cohérence entre plusieurs GPUs et CPUs.

Au moment de l'accès à une variable, notre structure de données interroge d'abord l'environnement d'exécution pour obtenir l'identifiant de l'unité qui demande l'accès. Ensuite, il vérifie si la version qu'est actuellement sur cette unité de calcul est valide. Dans le cas positif, il renvoie un pointeur vers la mémoire contenant la donnée. Si la version locale n'est pas valide, une copie d'une version valide est requise. Cette copie peut arriver, par exemple, quand on accède une variable pour la première fois, ou quand une autre unité de calcul a modifié la donnée.

Étant donné que les copies directes entre les GPUs ne sont pas supportés, les données doivent d'abord transiter par la mémoire principale, dans l'espace d'adressage du CPU. Notre interface d'accès aux données s'occupe de ces transferts, mais ils sont nettement chers et doivent être évités autant que possible.

11.3.2 Lancement de Calcul sur GPU

Dans un scénario multi-GPU traditionnel, le GPU où le calcul sera lancé est explicite dans le code. Ceci est contraignant dans notre contexte, vu que notre ordonnancer doit réaffecter le calcul a un GPU différent sans avoir a modifier le code. Pour ce faire, nous avons réimplémenté une partie de l'API CUDA Runtime. Le code est compilé comme d'habitude dans un scénario avec un seul GPU. Au moment de l'exécution notre implémentation de l'API CUDA intercepte les appels à la API CUDA. Quand un calcul CUDA est lancée, notre bibliothèque interroge notre ordonnanceur pour connaître le GPU cible. Ensuite, le contexte d'exécution est redirectioné vers un GPU différent, et ensuite le calcul est lancé. Une fois le calcul terminé, le contexte d'exécution est libérée pour que d'autres threads puissent l'accéder.

11.3.3 Tâches avec Implémentations Multiples

L'un des objectifs de notre interface est d'exécuter le calcul de façon transparente, que ce soit sur CPU ou sur GPU. Cela nécessite une interface pour cacher la implémentation des tâches, car les implémentations peuvent être très différents si elle vise un CPU ou un GPU. Nous proposons une interface d'haut niveau pour spécifier une tâche multi-implémentation s(Fig. 11.2). Tout d'abord une tâche est associée à une signature qui doit être respectée par toutes les implémentations. Cette signature inclut les paramètres de la tâche et leur mode d'accès (lecture ou écriture). Cette information sera en suite utilisée pour calculer les dépendances de données entre les tâches. Chaque tâche est codé comme un

objet foncteur. Il existe donc une claire séparation entre la définition d'une tâche et ses implémentation dans les différentes architectures. Notez que nous nous attendons à ce qu'au moins une implémentation soit fourni.

11.4 Ordonnancement Multi-GPU

Nous combinons deux approches pour l'ordonnancement des tâches. Nous nous appuyons d'abord sur le partitionnement de tâches qui est exécuté à chaque changement de graphe de tâches, par exemple quand des nouvelles collisions sont crées ou quand l'utilisateur interagit avec la simulation.. Entre deux partitionnements, le vol de travail est utilisé pour réduire le déséquilibre de charge qui peut se produire en raison du comportement dynamique de la simulation.

11.4.1 Partitionnement et Affectation des Tâches

Comme le partitionnement est fait lors de l'exécution de la simulation, il est important de réduire son coût au minimum. Le graphe de tâches est partitionné d'une manière simplifié en créant une partition par objet physique. Les tâches d'interaction, qui sont les tâches qui accèdent les deux objets, sont associées à l'un de ces objets. En suite, en utilisant des partitionneurs comme METIS ou SCOTCH, nous calculons un *mapping* entre des partitions et des unités de calcul qui minimise les communications entre les unités de calcule. Chaque fois que le graphe de tâches change en fonction des additions ou suppressions d'interactions entre les objets, le partitionnement est recalculée.

Associer l'ensemble des tâches qui partagent le même objet physique dans la même partition permet d'augmenter l'affinité entre ces tâches. Cela réduit considérablement les transferts de mémoire et améliore les performances en particulier sur les GPU, où ces transferts sont coûteux.

La simulation physique se caractérise aussi par un niveau élevé de localité temporelle, c'est à dire les changements d'une itération à la suivante sont généralement limités. Le vol de travail peut déplacer des partitions pour réduire le déséquilibre de charge. Ces mouvements ont une grande chance d'être utilisés pour la prochaine itération. Ainsi, si aucun nouveau partitionnement est nécessaire, chaque unité de calcul réutilise les partitions exécutées au cours de l'itération précédente.

11.4.2 Équilibrage de Charge Dynamique

Au début d'une nouvelle itération chaque unité de calcul a une liste de partitions à exécuter. L'exécution est alors contrôlé par l'algorithme de vol de travail de KAAPI [**?**]. Les unités de calcul commencent par rechercher des partitions prêtes dans leur file d'attente locale. Nous considérons une partition comme prête, uniquement si tous les données d'entrée nécessaires pour exécuter la partition sont prêts. S'il n'y a pas de partition prête dans la file d'attente locale, l'unité de calcul est considéré comme inactive et essaie de voler du travail d'une autres unité de calcul choisie au hasard.

Pour améliorer les performances de la simulation physique nous avons besoin de guider vol de travail pour à favoriser le regroupement d'objets en interaction sur une même unité de calcul. Nous utilisons une liste d'affinité de unités de calcul attachés à chacune des partitions : une partition possède une unité de

calcul dans sa liste de affinité uniquement si si cette unité contient au moins une tâche qui interagit avec la partition. Une unité de calcul est autorisé a voler une partition que si elle est dans la liste d'affinité de la partition. Différemment à l'approche utilisé par [6], ce contrôle d'affinité n'est utilisé que si la première tâche de la partition est déjà exécuté. Avant cela, n'importe quel processeur peut voler la partition.

11.4.3 Harnessing Multiple GPUs and CPUs

Nous ciblons les architecture composés de plusieurs processeurs et de GPUs. Dans cette architecture le temps de calcul d'une tâche dépend de l'unité de calcul, mais aussi de la nature de la tâche. Certaines taches sont plus adaptés aux CPU, tandis que d'autres s'exécutent plus rapidement sur GPU. Habituellement les GPU sont plus efficaces que les CPUs sur des tâches plus coûteuses avec un haut degré de parallélisme de données, et les processeurs sont généralement plus performants sur le problèmes de petite taille en raison du coût de transfert de données.

En se basant sur les travaux de [28], nous avons changé la politique de vol de travail pour attribuer les taches les plus lourdes sur l'unité le calcul la plus performante. Pendant l'exécution nous stockons le temps de calcul de chaque partition sur CPU et GPU. Les premières itérations sont utilisés comme une phase de réchauffement pour obtenir ces temps d'exécution. Ne pas avoir le meilleur rendement possible pour ces premières itérations est acceptable pour des simulations interactives qui fonctionnent habituellement pendant plusieurs minutes.

Au lieu d'avoir une file d'attente de partitions triées par leur temps d'exécution, nous utilisons un algorithme de seuil dynamique qui permet une meilleure exécution parallèle. Les partitions avec un rapport $\frac{Temps\ CPU}{Temps\ GPU}$ en dessous du seuil sont exécutés sur CPU, sinon sur GPU. Quand un voleur sélectionne au hasard une victime, il vérifie si la victime a une partition prête qui satisfait aux critères de seuil et il la vole. Sinon, l'unité de calcul choisit une nouvelle victime. Pour éviter qu'une unité de calcul reste inactive pendant longtemps, le seuil est augmenté chaque fois qu'un processeur ne parvient pas à voler, et diminue à chaque fois un GPU échoue.

11.5 Résultats

Pour valider notre approche, nous avons utilisé différentes scénarios de simulation, y compris des objets indépendants ou en collisions et des objets attachés. Les test ont été fait sur un quad-core Intel Nehalem 3GHz avec 4 Nvidia GeForce GTX 295 GPU dual. Les tests utilisant 4 GPU ont été effectués sur un double quad-core Intel Nehalem à 2,4 GHz avec 2 Nvidia GeForce GTX 295 GPU dual. Les résultats présentés sont obtenus à partir de la valeur moyenne de 100 exécutions.

11.5.1 Objets en Collision

La première scène est composé de 64 objets déformables qui tombent avec la gravité ((Fig. 11.4). Cette scène est homogène et tous les objets sont composés de 3k particules, simulés en utilisant une méthode des éléments finis avec un gradient conjugué comme solveur d'équations. Au début de la simulation tous les objets sont séparés, puis le nombre de collisions augmente pour atteindre 60 paires de

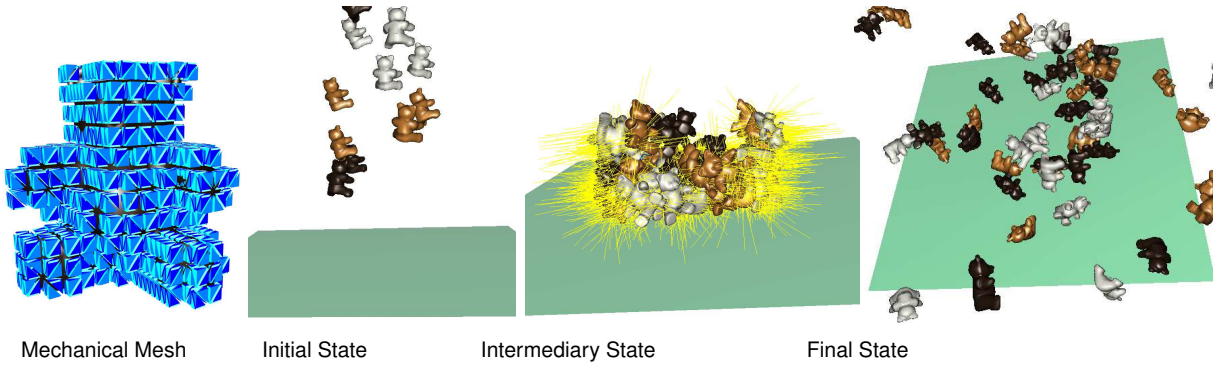


Figure 11.3: Simulation de 64 objets avec des collisions. Chaque objet est déformable simulé à l'aide d'une méthode des éléments finis (FEM) avec 3k particules.

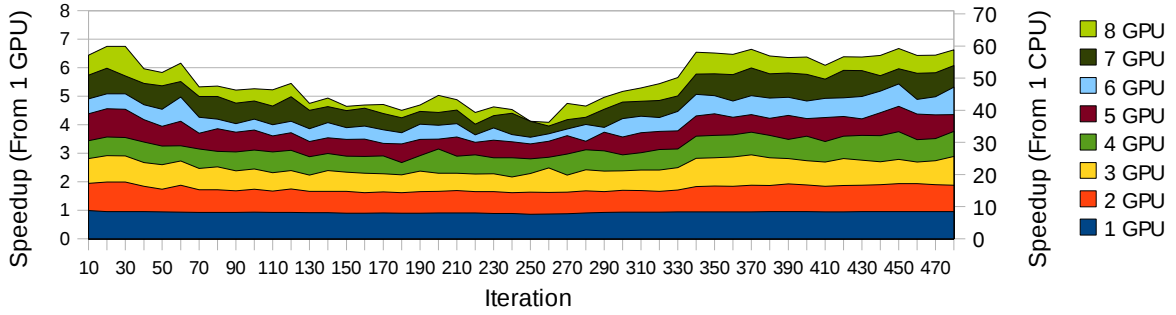
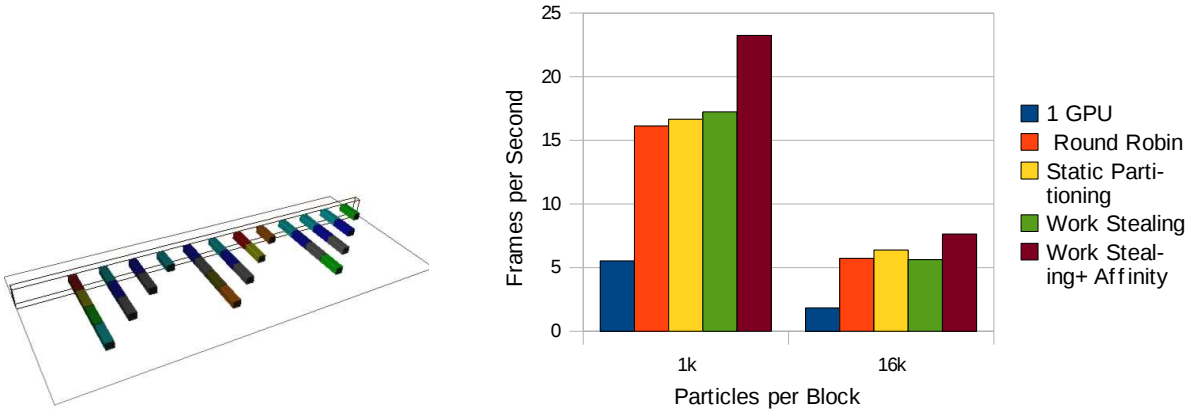


Figure 11.4: Accélération par itération lors de la simulation 64 objets déformables (Fig. 11.3) en utilisant jusqu'à 8 GPUs

collision avant que les objets commencent à se séparer de nouveau sous l'action des forces de répulsion. La référence de temps CPU séquentiel est de 3.8s par itération.

Nous pouvons observer que lorsque les objets ne sont pas en collision (début et fin de la simulation) l'accélération (par rapport à un GPU) est proche de 7 à 8 GPU. Comme prévu la cadence diminue à mesure que le nombre de collisions augmente, mais nous pouvons encore obtenir au moins 50 % de rendement (à l'itération 260). Au cours de nos expériences, nous avons observé une forte variance du temps d'exécution à l'itération suivante de l'apparition de nouvelles collisions. Cela est dû à l'augmentation du nombre vol nécessaires pour adapter la charge de la partition. Le vol a des coûts car il déclenche des transferts de mémoire GPU-CPU-GPU.

La deuxième scène testé est très similaire, nous avons uniquement changé le modèle mécanique des objets pour obtenir une scène composée d'objets hétérogènes. La moitié des 64 objets est simulée en utilisant un méthode des éléments finis, tandis que le reste utilise un modèle de masses-ressorts. Les tailles d'objet est elle aussi hétérogène, allant de 100 à 3k particules. Nous avons obtenu une accélération moyenne de 4,4, à comparer avec 5,3 obtenue pour la scène homogène (Fig. 11.4). Cette accélération est plus faible en raison de la difficulté plus élevée de trouver une répartition équilibrée en raison de l'hétérogénéité de la scène.



(a) (b)

Figure 11.5: (a)Un ensemble de barres souples fixées à un mur (chaque couleur est associée un bloc qui compose une barre). (b) Performances avec des blocs de tailles différentes, en utilisant différentes stratégies d'ordonnancement

11.5.2 Vol de Travail Guidé par Affinité

Nous avons étudié l'efficacité de notre approche de vol de travail guidé par l'affinité . Nous avons simulé 30 objets mous regroupés en 12 barres (Fig. 11.5(a)). Ces barres sont placés horizontalement et sont attachés à un mur. Elles fléchissent sous l'action de la pesanteur. Les blocs liés en une seule barre sont en interaction similaire à une collision entre objets. Nous comparons ensuite les résultats de cette simulation, en activant des différentes stratégies d'ordonnancement (Fig. 11.5(b)). La première stratégie attribue des blocs à des 4 GPU d'une manière circulaire (*Round-robin*). Le résultat est une distribution qui a une bonne équilibrage de charges, mais une pauvre localité des données, puisque les blocs de une barre sont situés dans des différents GPUs. La deuxième stratégie utilise un partitionnement statique, qui regroupe les blocs dans une même barre sur le même GPU. Cette solution a une bonne localité des données puisque aucun transfert des données entre différents GPU est fait, mais la charge de travail n'est pas bien équilibrée vu que les bars ont un nombre différent de blocs. La troisième stratégie s'appuie sur un vol de travail traditionnel. Elle surpasse légèrement le partitionnement statique car elle assure un meilleur équilibrage de charge. Il est aussi plus performant que la distribution Round-robin parce que un des GPUs est un peu plus chargé que les autres car il exécute le code OpenGL pour le rendu de la scène sur un écran. Pour les objets plus grands, le coût des transferts de mémoire du vol de travail devient plus important, rendant le vol de travail moins efficace que les 2 autres méthodes. Lorsqu'ils est guidé par l'affinité, le vol de travail donne les meilleurs résultats pour les deux tailles de blocs. Il permet de réaliser une bonne répartition des charges, tout en préservant la localité des données.

11.5.3 Tests avec CPUs et GPUs

Nous avons effectué des tests combinant plusieurs GPU et CPU sur une machine avec 4 GPU et 8 cores. Comme les GPU sont des dispositifs passifs, un core est associée à chaque GPU pour le contrôler. Nous avons que 4 cores qui peuvent entre utilisés pour le calcul de la simulation. La scène se compose d'objets indépendants avec 512 à 3 000 particules. Nous comparons ensuite le vol de travail traditionnel avec le vol de travail guidé par priorité (Sec. 11.4.3).

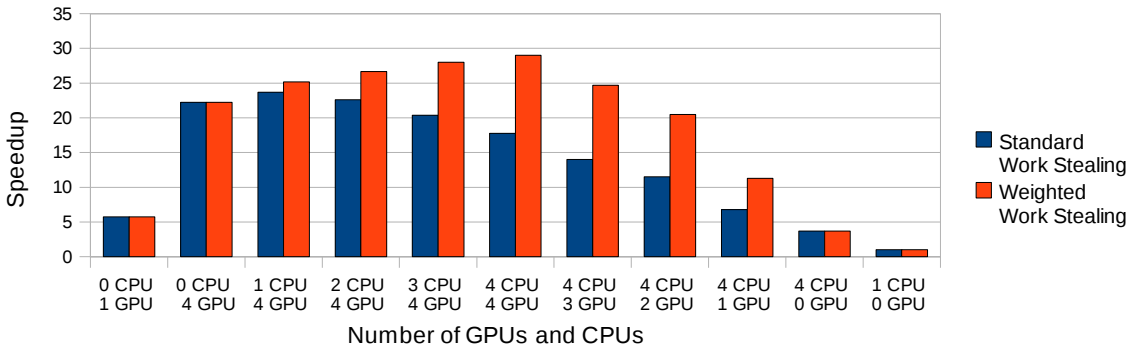


Figure 11.6: Tests de performances avec des diverses combinaisons CPU et GPU.

Les résultats (Fig. 11.6) montrent que notre vol de travail guidé par priorités est toujours plus performant que le vol de travail traditionnel dès que au moins un CPU et un GPU soient impliqués dans le calcul. Nous avons aussi des " accélérations de coopératives". Par exemple, l'accélération avec 4 CPUs et 4 GPUs(29), est plus grand que la somme des accélérations des 4 processeurs (3,5) et des 4 GPUs (22). La raison vient de l'architecture SIMD des GPU. Le traitement d'un petit objet prend parfois autant de temps qu'un grand. Avec les vol de travail guidé par priorité, le CPU exécute des tâches qui ne sont pas bien adaptés aux GPU. Ensuite, le GPU ne traitera que des tâches plus importantes, résultant des gains de performance supérieurs.

De son coté le vol de travail standard conduit à des desaccélérations " concurrentielles". La simulation est plus lente avec 4 GPU et 4 CPU que avec seulement 4 GPU. Il peut être expliqué par le fait que quand un CPU a une tâche qui n'est pas bien adapté à son architecture, il peut devenir le chemin critique de l'itération, puisque les tâches ne peuvent pas être préemptées.

11.6 Conclusion

Dans ce chapitre nous avons proposé de combiner le partitionnement et le vol de travail pour paralléliser les simulations physique sur plusieurs GPU et CPU. Nous essayons de profiter de la localité spatiale et temporelle pour l'ordonnancement. La localité temporelle repose principalement sur la réutilisation de la distribution de partition entre les itérations consécutives. La localité spatiale est appliquée pour guide le vol à choisir des partitions qui interagissent avec celles déjà présentes dans le processeur. En outre, dans le contexte hétérogène où les deux CPU et GPU sont considérés, nous utilisons un vol de travail guidé par priorité qui sert à favoriser l'exécution des partitions de faible poids sur les processeurs et les tâches ls plus lourdes sur les GPUs. L'objectif est de donner à chaque unité de calcul les partitions les plus adaptés a son mode de calcul. Les expériences confirment les avantages de ces stratégies. En particulier, nous obtenons des accélérations "coopératives"quand CPU et GPU sont considérés dans une même simulation.

Chapter 12

Conclusion

12.1 Objectifs

Au cours de cette thèse nous avons étudié l'exécution de la simulation physique dans le but d'améliorer l'interactivité. Ce type d'application implique un important travail de collaboration avec un ensemble d'algorithmes de natures différentes qu'interagissent pour produire une seule scène. Faire en sorte qu'une application supporte le travail collaboratif nécessite un effort important de génie logiciel.

Le framework Sofa propose une interface bien définie pour les composants de simulation physique, permettant une intégration transparente de nouveaux algorithmes. L'objectif de cette thèse était d'améliorer l'interactivité physique soit en proposant de nouveaux algorithmes de simulation physique ou en utilisant des architectures parallèles. En outre, la solution proposée doit avoir un impact réduit dans le code, de sorte qu'il soit accessible pour les collaborateurs qui ne sont pas experts dans le parallélisme

L'interface bien définie de Sofa facilite l'isolement de l'environnement parallèle à partir du code physique permettant d'apporter des architectures parallèles accessibles aux développeurs de simulation physique. L'architecture cible initiale ce sont les machines à mémoire partagée multi-core. Nous avons également traité le cas des processeurs spécialisés comme les GPU, quand une version spécialisée d'un algorithme est disponible.

12.2 Méthodologie et Contributions

Nous avons commencé nos travaux par une évaluation des algorithmes de simulation physique afin de mieux identifier les enjeux de ce type d'application. À la suite de ce travail, nous avons remarqué un manque algorithme de détection de collision en temps discrets qui sont robustes avec des grands pas de temps. Cette absence réduit l'efficacité des scènes où les objets sont profondément interpénétrées d'une étape à l'autre. Notre première contribution a été de proposer une **détection de collision par lancer de rayons**, qui est capable de séparer les objets interpénétrés.

Les travaux sur le parallélisme ont commencé avec un prototype de parallélisation à gros grain de cet algorithme de détection de collision utilisant KAAPI, un environnement parallèle développé dans

l'équipe MOAIS. Dans cette approche, les boîtes englobantes sont calculés en parallèle. En utilisant ces boîtes englobantes les paires d'objets sont calculées de façon indépendante. La principale difficulté provient du déséquilibre de charge en raison des différences de taille entre les objets. Considérant le pipeline physique, nous avons observé que la partie la plus complexe à paralléliser a été l'étape d'intégration du temps. La détection de collision implique généralement une seule méthode qui est utilisée sur tous les objets. De l'autre côté, lors de l'intégration du temps, différents algorithmes sont utilisés pour calculer l'état interne de chaque objet, qui peut conduire à plus de dépendances complexes entre les tâches.

Nous avons développé une version parallèle du framework SOFA pour les architectures multi-core. Nous avons adapté KAAPI afin de mieux s'adapter aux flux d'exécution de la simulation physique. Nous avons d'abord extrait un graphe de tâches qui est partitionné et déployé sur les processeurs. Une contribution de ce travail était **l'amélioration de la localité** par le regroupement des tâches qui modifient les mêmes des données dans une même partition et aussi par le redéploiement de une nouvelle itération basée sur le placement des données de la itération précédente. Une autre contribution a été les **boucles dynamiques** exprimées dans le graphe de tâches. Ils ont permis de simuler des algorithmes plus complexes comme me gradient conjugué qui nécessite une boucle de convergence. Aussi KAAPI a une stratégie d'équilibrage de charge basé sur le vol de travail, qui permets de corriger le déséquilibre de distribution de charge entre les processeurs.

Les résultats de la parallélisation multi-core montrent que nous pouvons obtenir une accélération presque-optimale dans les cas où les objets ne sont pas en collision. Nous avons également observé une efficacité de près de 50% lorsque les objets sont en collision. Nos extensions à KAAPI, a permis de mieux exposer le parallélisme de la simulation physique, et l'analyse des données de dépendance a permis d'éviter la synchronisation entre les threads. Toutes ces améliorations de performance ont été obtenus sans aucun changement dans les algorithmes physiques, puisque nous avons exploité le parallélisme entre les différents objets simulés. De nouveaux algorithmes qui sont intégrés dans SOFA, qui suivent l'interface de ce framework peuvent profiter des architecture parallèle sans modifier le code.

Au même temps que les architectures multi-core sont devenus populaires, les processeurs graphiques programmables se sont émergé comme une alternative pour le calcul des algorithmes du type data-parallèle.. La simulation physique, tels que les objets déformables, peuvent généralement être implémentés en utilisant une approche data-parallèle. Par exemple, les particules d'un objet peuvent être calculés en parallèle en exécutant la même opération sur chacun d'eux.

Pour profiter pleinement de machines composé de GPU et CPU, nous avons étendu nos travaux sur les architectures multi-core pour le contexte des architectures hybrides composées par multi-processeurs et multi-GPU. Nous avons d'abord proposé une interface pour créer des **tâches multi-architecture** de sorte qu'un ordonnanceur puisse choisir d'exécuter la implémentation: CPU ou GPU. Cette interface étend l'interface Athapascan qui a été utilisé dans la parallélisation multi-core. L'application directe de nos travaux sur les architectures multi-core sur des architectures hybrides ne produisent pas de résultats satisfaisants. De nouvelles contraintes doivent être considéré; comme le transfert de mémoire entre CPU et GPU, et le choix de la tâche qui est mieux adaptée à chaque architecture.

Une contribution de ce travail sur des architectures hybrides était de **guider le vol de travail** pour mieux choisir les tâches victime. Tout d'abord, pour réduire le transfert de la mémoire entre les unités de calcul, nous guidons le vol pour respecter la disposition de la scène, par exemple en attribuant une **affinité** entre les objets qui sont en collision. Un processeur inactif donnera préférence à des tâches qui correspondent à des objets physiques qui interagissent avec les objets exécutés par le voleur.

Deuxièmement, nous utilisons un **vol de travail guidé par priorité** pour choisir les tâches les plus adaptés une unité de traitement. Il favorise l'exécution des partitions de faible poids sur les processeurs et celles des poids plus important sur les GPU. Le poids peut être calculé en fonction de la taille du corps, ou du temps d'exécution obtenu à l'exécution. L'objectif d'un vol de tache basé sur le poids des objets est de donner à chaque unité de calcul les partitions qu'il exécute le plus efficacement.

Les expériences montrent que nous pouvons obtenir une accélération d'environ 4 lorsque l'aide de 8 GPU. Aussi nous sommes parvenus à obtenir des accélérations coopératives lors que CPU et GPU sont combinés dans une même scène, c'est à dire nous avons obtenu une accélération plus élevée que la somme de ce qu'est obtenu par chaque type d'unités de calcul.

12.3 Travaux Futurs

Les contributions de cette thèse ont crée les bases et validé des nombreux concepts pour une simulation physique parallèle à l'aide des machines multi-CPU et multi-GPU. Au même temps elle ouvre l'étude des nouvelles possibilités d'améliorations. Dans cette section, nous allons détailler quelques-unes des questions permettant de mieux exploiter les machines parallèles pour la simulation physique.

12.3.1 Détection de Collision Basé sur la Localité des Données

La détection de collision sur GPU a améliorée de façon significative la performance. Il peut être facilement perçu lorsque l'on compare une simulation avec un seul processeur et une simulation avec un seul GPU. Cependant, avec l'intégration du temps multi-GPU, l'utilisation d'un seul GPU pour la détection de collision devient un goulot d'étranglement. L'ajout d'unités de calcul ne fait que augmenter les performances du temps d'intégration, sans aucune amélioration pour la détection de collision.

La parallélisation multi-GPU de la détection de collision peut atténuer ce goulot d'étranglement. Mais la parallélisation seule ne peut pas garantir une meilleure performance. Une des faiblesses de notre approche consiste à considérer l'intégration du temps et de détection de collision comme des étapes de calculs complètement distinctes. Toutefois, ces deux étapes ont besoin de communiquer à chaque pas de temps, la détection de collision a besoin de connaître les nouvelles positions de l'objet calculés par l'étape d'intégration du temps. Une approche multi-GPU pour la détection de collision devrait réduire les transferts de mémoire, parce que la localité des données est essentielle pour obtenir de bonnes performances en utilisant plusieurs GPUs.

Une façon de minimiser le transfert lors du passage de l'intégration du temps à la détection de collision est d'utiliser l'emplacement des données à la fin de l'étape d'intégration du temps pour déterminer où la détection de collision sera calculée. La parallélisation multi-GPU de l'intégration du temps proposée au cours de cette thèse réduit les transferts de mémoire en guidant le vol de travail pour faire en sorte qu'il groupe les objets en collision dans un même processeur. On peut alors calculer la détection de collision des corps en collision dans la même unité de calcul que celle de l'intégration du temps. La *broad phase* de chaque objet peut être calculé sur place, dans un GPU qui contient déjà les données de l'objet. Ensuite, les boîtes englobantes sont transférés au CPU pour calculer les groupes de collision. Chaque groupe de collision est alors attribué à un GPU, en conservant la localité des objets. Habituellement, le nombre de nouvelles collisions crées d'un itération à l'autre est faible. Cela signifie que uniquement un

petit nombre d'objets devra être transféré lors de la détection de collision.

12.3.2 Réduire de transfert de données dans les interactions

Comme nous avons présenté ci-dessus, nous pouvons réduire le transfert de mémoire entre les GPU en préservant la localité des données lorsqu'il s'agit de décider où exécuter la détection de collision. Toutefois, dans certains cas, nous ne pouvons pas éviter de transférer des données entre les GPU. Par exemple quand un corps est entré en collision avec des objets dont les temps d'intégration se fait dans des unités de calcul différents. Dans ce cas, deux unités ou plus accèdent les données d'un même objet.

Une façon de réduire le coût de transfert est en limitant la quantité de données transférées. Dans une simulation multi-modèle, un objet peut avoir plusieurs représentations différentes et ces représentations peuvent avoir différents niveaux de détail. Par exemple, nous pouvons avoir un modèle mécanique qui est plus grossier que le modèle de collision. La même chose peut se produire avec le modèle visuel. Sofa a déjà la plupart des bases de ce type d'optimisation. Transférer uniquement le modèle le plus grossier et mettre à jour le plus fin localement sur chaque GPU peut réduire la quantité de données transférées spécialement quand il ya une énorme différence de taille entre les modèles.

12.3.3 Décomposition de Domaine Dynamique

Le grain de nos tâches parallèles sont directement liées au nombre de degrés de liberté dans un objet. Une tâche représente une opération sur tout le corps. Le fait de tomber sur un objet de grande taille peut influencer sur la performance globale. L'origine de cet inconvénient est que, au moment de l'exécution, il n'existe aucun mécanisme pour décomposer le calcul d'un objet.

Le parallélisme à grain fin, comme celui utilisé pour la parallélisation du GPU, peut accélérer le calcul d'un corps. Par exemple, nous pouvons réduire le grain de calcul en créant des nouvelles tâches à l'intérieur d'une plus grande. Cette solution nécessite une modification du fonctionnement interne des algorithmes physique. Une autre façon de décomposer le calcul est de décomposer un corps en sous-parties. C'est le cas de simulation de masse-ressort de la Section **??**, où les différentes parties d'un tissu sont attachés pour assurer la conformité avec la simulation originale. Une décomposition de domaine similaires pourrait être employées à l'exécution lorsque le simulateur physique détecte l'existence d'un objet qui affectent la performance.

12.3.4 Environnement Parallèle

Même si notre architecture cible est machines à mémoire partagée, les concepts de ce travail ont été implémentés en utilisant un environnement parallèle qui supporte les architectures à mémoire distribuée (KAAPI). En conséquence, la contribution spécifique de parallélisme de cette thèse pourrait éventuellement être appliquée à un modèle à mémoire distribuée. Pour ce faire, il faudra, bien sûr, un effort supplémentaire des développeurs des composants, pour développer des codes qui soient compatibles avec une architecture à mémoire distribuée. Aussi la gestion de la boucle conditionnelle doit être adaptée à un contexte mémoire distribuée afin d'assurer la cohérence entre toutes les instances d'une boucle.

Au cours de nos travaux sur les architectures hybrides, nous avons utilisé partitionneurs extérieur pour obtenir une répartition de travail initial qui minimise la communication. La bibliothèque SCOTCH est capable de calculer une répartition en tenant compte des architectures hétérogènes, par contre nous n'avons pas exploité cette fonctionnalité, et les unités de traitement ont été considérés comme étant homogènes. Une possible alternative serait de trouver des paramètres pour décrire une architecture composée de GPU et CPU et utiliser SCOTCH pour attribue la bonne quantité de travail à la bonne architecture basée sur les informations obtenues lors de l'exécution.

Une autre faiblesse de notre implémentation multi-GPU est d'occuper un processeur pour chaque GPU. Il limite l'exploitation de toute la puissance disponible de l'architecture, un processeur consacré au contrôle d'un GPU est inactif s la plupart du temps. Cet inconvénient peut être contourné en déplaçant le contrôle du vol de travail à l'intérieur du GPU de façon à pouvoir obtenir plus de travail en cas d'inactivité sans dépendre d'un CPU. Avec la convergence entre les CPUs et GPUs, nous nous attendons aussi d'avoir une architecture où les transferts de mémoire ne sont pas si coûteux que maintenant, principalement par l'intégration de processeurs spécialisés dans la carte mère capables d'accéder à la mémoire principale. Cette facilité de communication permettrait de implémenter un vol de travail plus efficace entre CPU et GPU en allant vers considérer le GPU comme un périphérique autonome et non plus passive.

Bibliography

[1] Bullet Physics Library. http://www.bulletphysics.com. 61

[2] Kaapi. http://kaapi.gforge.inria.fr/. 63

[3] Nvidia PhysX. http://www.nvidia.com/physx. 61

[4] Open MASK. http://www.irisa.fr/bunraku/OpenMASK/. 58

[5] Simulation open framework architecture. http://www.sofa-framework.org. 21

[6] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM. 88, 89, 133

[7] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of High-Performance Graphics 2009*, 2009. 32

[8] M. Aldinucci, M. Torquati, and M. Meneghin. FastFlow: Efficient Parallel Streaming Applications on Multi-core. *ArXiv e-prints*, September 2009. 44

[9] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. Flowvr: a middleware for large scale virtual reality applications. In *Proceedings of Euro-par 2004*, Pisa, Italia, August 2004. 58

[10] J. Allard and B. Raffin. Distributed physical based simulations for large vr applications. *Virtual Reality Conference, 2006*, pages 89–96, March 2006. 15, 60

[11] Jérémie Allard, Stéphane Cotin, François Faure, Pierre-Jean Bensoussan, François Poyer, Christian Duriez, Hervé Delingette, and Laurent Grisoni. Sofa - an open source framework for medical simulation. In *Medicine Meets Virtual Reality (MMVR'15)*, Long Beach, USA, February 2007. 16, 28, 114

[12] Jérémie Allard, Stéphane Cotin, François Faure, Pierre-Jean Bensoussan, François Poyer, Christian Duriez, Hervé Delingette, and Laurent Grisoni. SOFA - an open source framework for medical simulation. In *Medicine Meets Virtual Reality (MMVR'15)*, pages 1–6, Long Beach, California, Etats-Unis, February 2007. 119

[13] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. Stapl: An adaptive, generic parallel c++ library. pages 195–210. 2003. 47

[14] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Timmie G. Smith, Gabriel Tanase, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger. Stapl: An adaptive, generic parallel c++ library. In Henry G. Dietz, editor, *LCPC*, volume 2624 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2001. 47

[15] Sylvester Arnab and Vinesh Raja. A deformable surface model with volume preserving springs. In *AMDO '08: Proceedings of the 5th international conference on Articulated Motion and Deformable Objects*, pages 259–268, Berlin, Heidelberg, 2008. Springer-Verlag. 9

[16] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. pages 201–262, 1989. 23

[17] Cédric Augonnet and Raymond Namyst. A unified runtime system for heterogeneous multicore architectures. In *Proceedings of the International Euro-Par Workshops 2008, HPPC'08*, volume 5415 of *Lecture Notes in Computer Science*, pages 174–183, Las Palmas de Gran Canaria, Spain, August 2008. Springer. 51

[18] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference, Lecture Notes in Computer Science*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874, Delft, The Netherlands, August 2009. Springer. 51

[19] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *Euro-Par'09*, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag. 52

[20] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. A proposal for task parallelism in openmp. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 1–12, Berlin, Heidelberg, 2008. Springer-Verlag. 45

[21] David A. Bader, Varun Kanade, and Kamesh Madduri. Swarm: A parallel programming framework for multicore processors. In *IPDPS*, pages 1–8. IEEE, 2007. 44

[22] P. Banks, P. Edwards, J.C. Rodriguez, R. Martin, M. Williams, S. Cebollero, and M. Halloran. Cape-open : Open interface specifications. Technical report, The CAPE-OPEN Laboratories Network, 2001. http://www.colan.org. 54, 72

[23] D. Baraff. Rigid body simulation. *SIGGRAPH Course Notes 1992*, 19. 9

[24] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 23–34, New York, NY, USA, 1994. ACM. 15

[25] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 43–54, New York, NY, USA, 1998. ACM. 12, 15

[26] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press. 41

[27] Jacek Bazewicz, Denis Trystram, Klaus Ecker, and Brigitte Plateau, editors. *Handbook on Parallel and Distributed Processing*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000. 39

[28] M. A. Bender and M. O. Rabin. Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk. *Theory of Computing Systems Special Issue on SPAA '00*, 35(3):289–304, 2000. 89, 133

[29] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. NAMD: A Portable and Highly Scalable Program for Biomolecular Simulations. Technical Report UIUCDCS-R-2009-3034, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2009. 60

[30] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. Technical report, Cambridge, MA, USA, 1996. 44, 45

[31] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 297–308, New York, NY, USA, 1996. ACM. 88

[32] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. 54

[33] David Blythe, Brad Grantham, Tom McReynolds, and Scott R. Nelson. Advanced Graphics Programming Techniques Using OpenGL. *SIGGRAPH '99 Course*, 1999. 41

[34] D. Bourguignon and M.P. Cani. Controlling anisotropy in mass-spring systems. In *Computer animation and simulation 2000: proceedings of the Eurographics Workshop in Interlaken, Switzerland, August 21-22, 2000*, page 113. Springer Verlag Wien, 2000. 9

[35] Robert Bridson, Ronald Fedkiw, and John Anderson. Robust treatment of collisions, contact and friction for cloth animation, 2002. 28, 114

[36] K. Brown, S. Attaway, S. Plimpton, and B. Hendrickson. Parallel strategies for crash and impact simulations. *Computer Methods in Applied Mechanics and Engineering*, 184(2):375–390, 2000. 64

[37] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. 44, 45

[38] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *In Proc. of ACM Interactive 3D Graphics Conference*, pages 189–196, 1995. 13

[39] Olivier Comas, Zeike A. Taylor, Jérémie Allard, Sébastien Ourselin, Stéphane Cotin, and Josh Passenger. Efficient nonlinear fem for soft tissue modelling and its gpu implementation within the open source framework sofa. In *ISBMS '08: Proceedings of the 4th international symposium on Biomedical Simulation*, pages 28–39, Berlin, Heidelberg, 2008. Springer-Verlag. 42

[40] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, Jan-Mar 1998. 52

[41] Gilles Debunne, Mathieu Desbrun, Marie-Paule Cani, and Alan H. Barr. Dynamic real-time deformations using space & time adaptive sampling. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 31–36, New York, NY, USA, 2001. ACM. 10

[42] Gilles Debunne, Mathieu Desbrun, Marie-Paule Cani, and Alan H. Barr. Dynamic real-time deformations using space and time adaptive sampling. In *Computer Graphics (ACM SIGGRAPH)*, Annual Conference Series. ACM Press / ACM SIGGRAPH, Aug 2001. Proceedings of SIGGRAPH'01. 11

[43] H. Delingette, S. Cotin, and N. Ayache. Efficient linear elastic models of soft tissues for real time surgery simulation. In *Medecine Meets Virtual Reality VII*, Interactive Technology and the New Paradigm for He althcare, pages 139–151. IOS Press, January 1999. 11

[44] Mathieu Desbrun, Peter Schröder, and Alan Barr. Interactive animation of structured deformable objects. In *Proceedings of the 1999 conference on Graphics interface '99*, pages 1–8, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. 12

[45] Romain Dolbeau, Stéphane Bihan, and Francóis Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *First Workshop on General Purpose Processing on Graphics Processing Unit*, 2007. 51

[46] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP Task Scheduling Strategies. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *Lecture Notes in Computer Science: Proceedings of the 4th International Workshop on OpenMP*, volume 5004, pages 100–110. Springer, Springer, May 2008. 45

[47] Thiago S. M. C. de Farias, Mozart W. S. Almeida, ao Marcelo X. N. Teixeira, Jo Veronica Teichrieb, and Judith Kelner. A high performance massively parallel approach for real time deformable body physics simulation. In *SBAC-PAD '08: Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*, pages 45–52, Washington, DC, USA, 2008. IEEE Computer Society. 60, 83

[48] François Faure, Sébastien Barbier, Jérémie Allard, and Florent Falipou. Image-based collision detection and response between arbitrary volumetric objects. In *ACM Siggraph/Eurographics Symposium on Computer Animation, SCA 2008, July, 2008*, Dublin, Irlande, July 2008. 15, 32, 64, 83

[49] Susan Fisher and Ming C. Lin. Deformed distance fields for simulation of non-penetrating flexible bodies. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 99–111, New York, NY, USA, 2001. Springer-Verlag New York, Inc. 14

[50] Michael J. Flynn and Kevin W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28(1):67–70, 1996. 38

[51] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998. http://supertech.csail.mit.edu/papers/cilk5.pdf. 83, 129

[52] A. Fuhrmann, G. Sobotka, and C. Groß. Distance fields for rapid collision detection in physically based modeling. In *Proceedings of GraphiCon 2003*, pages 58–65. Citeseer, 2003. 14

[53] A. Fuhrmann, G. Sobottka, and C. Groß. Distance fields for rapid collision detection in physically based modeling. In *Proceedings of GraphiCon 2003*, pages 58–65, Moscow, September 2003. 14

[54] Franćois Galilée, Jean-Louis Roch, Gerson G.H. Cavalheiro, and Mathias Doreille. Athapascan-1: On-line building data flow graph in a parallel language. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:88, 1998. 47

[55] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23, New York, NY, USA, 2007. ACM. 29, 65, 88, 114

[56] Joachim Georgii, Florian Echtler, and Rüdiger Westermann. Interactive simulation of deformable bodies on gpus. In *Proceedings of Simulation and Visualisation 2005*, pages 247–258, 2005. 60, 83

[57] Joachim Georgii and Rüdiger Westermann. Mass-spring systems on the gpu. *Simulation Modelling Practice and Theory*, 13:693–702, 2005. 9

[58] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: a hierarchical structure for rapid interference detection. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180, New York, NY, USA, 1996. ACM. 13

[59] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, 1996. 43

[60] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994. 43

[61] Michael Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006. ACM. 40

[62] Eladio Gutiérrez, Sergio Romero, Luis F. Romero, Oscar Plata, and Emilio L. Zapata. Parallel techniques in irregular codes: cloth simulation as case of study. *J. Parallel Distrib. Comput.*, 65(4):424–436, 2005. 60

[63] Mark Harris. Fast fluid dynamics simulation on the gpu. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 220, New York, NY, USA, 2005. ACM. 42

[64] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. 42

[65] Michael Hauth, Olaf Etzmuss, and Universität Tübingen. A high performance solver for the animation of deformable objects using advanced numerical methods. In *In Proc. Eurographics 2001 (2001), Chalmers A., Rhyne T.-M., (Eds*, pages 319–328, 2001. 12

[66] Everton Hermann, François Faure, and Bruno Raffin. Ray-traced collision detection for deformable bodies. In *3rd International Conference on Computer Graphics Theory and Applications, GRAPP 2008, January, 2008*, Funchal, Madeira, Portugal, January 2008. 21

[67] Everton Hermann, Bruno Raffin, and François Faure. Interactive physical simulation on multicore architectures. In *Eurographics Symposium on Parallel and Graphics and Visualization, EGPGV'09, March, 2009*, Munich, 2009. 63

[68] Min Hong, Sunhwa Jung, Min-Hyung Choi, and Samuel W. J. Welch. Fast volume preservation for a mass-spring system. *IEEE Comput. Graph. Appl.*, 26(5):83–91, 2006. 9

[69] Donald House, Richard W. Devaul, and David E. Breen. Towards simulating cloth dynamics using interacting particles. *International Journal of Clothing Science and Technology*, 8:75–94, 1996. 9

[70] Chao Huang and Laxmikant V. Kale. Charisma: Orchestrating migratable parallel objects. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007. 60

[71] Christopher J. Hughes, Radek Grzeszczuk, Eftychios Sifakis, Daehyun Kim, Sanjeev Kumar, Andrew P. Selle, Jatin Chhugani, Matthew Holliman, and Yen-Kuang Chen. Physical simulation for animation and visual effects: parallelization and characterization for chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):220–231, 2007. 60

[72] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, 1989. 55

[73] Kai Hwang. *Advanced Computer Architecture: Parallelism,Scalability,Programmability*. McGraw-Hill Higher Education, 1992. 39

[74] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. *SIGOPS Oper. Syst. Rev.*, 21(5):77–93, 1987. 59

[75] Lenka Jerabkova, Christian Terboven, Samuel Sarholz, Torsten Kuhlen, and Christian Bischof. Exploiting multicore architectures for physically based simulation of deformable objects in virtual environments. In *Virtuelle und Erweiterte Realität, 4. Workshop der GI-Fachgruppe VR/AR, Weimar, Germany*, 2007. 60

[76] J. E. Jones. On the determination of molecular fields. ii. from the equation of state of a gas. *Proceedings of the Royal Society of London. Series A*, 106(738):463–477, October 1924. 9

[77] M. Joselli, E. Clua, A. Montenegro, A. Conci, and P. Pagliosa. A new physics engine with automatic process distribution between CPU-GPU. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 149–156. ACM, 2008. 61

[78] L. V. Kalé, Mark Hills, and Chao Huang. An orchestration language for parallel objects. In *Proceedings of Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR 04)*, Houston, Texas, October 2004. 60

[79] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996. 51, 59, 60

[80] George Karypis and Vipin Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995. 50, 55

[81] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998. 68

[82] James T. Klosowski, Martin, Held Joseph, S. B. Mitchell, Ãœ Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4:21–36, 1998. 13

[83] S. Kumar, C. Huang, G. Zheng, E. Bohm, A. Bhatele, J. C. Phillips, H. Yu, and L. V. Kalé. Scalable molecular dynamics with namd on the ibm blue gene/l system. *IBM J. Res. Dev.*, 52(1/2):177–188, 2008. 60

[84] J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra. The playstation 3 for high-performance scientific computing. *Computing in Science & Engineering*, 10(3):84–87, May-June 2008. 41

[85] Monica S. Lam and Martin C. Rinard. Coarse-grain parallel programming in jade. *SIGPLAN Not.*, 26(7):94–105, 1991. 47

[86] Orion Sky Lawlor and Laxmikant V. Kalée. A voxel-based parallel collision detection algorithm. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 285–293, New York, NY, USA, 2002. ACM. 64

[87] Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors. *Euro-Par '97 Parallel Processing, Third International Euro-Par Conference, Passau, Germany, August 26-29, 1997, Proceedings*, volume 1300 of *Lecture Notes in Computer Science*. Springer, 1997. 47

[88] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March-April 2008. 41

[89] Kevin McManus, Mark Cross, Chris Walshaw, Nick Croft, and Alison Williams. Parallel performance in multi-physics simulation. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, pages 806–815, London, UK, 2002. Springer-Verlag. 60

[90] Brian Mirtich. Timewarp rigid body simulation. In *Proc. of ACM SIGGRAPH*, pages 193–200, 2000. 59

[91] Brian Mirtich and John Canny. Impulse-based simulation of rigid bodies. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 181–ff., New York, NY, USA, 1995. ACM. 15

[92] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997. 24, 112

[93] J.J. Monaghan. Smoothed Particle Hydrodynamics. *ann*, 30:543–74, 1992. 9

[94] J. Montrym and H. Moreton. The geforce 6800. *IEEE Micro*, 25(2):41–51, March-April 2005. 41

[95] Matthias Müller, Julie Dorsey, Leonard McMillan, Robert Jagnow, and Barbara Cutler. Stable real-time deformations. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 49–54, New York, NY, USA, 2002. ACM. 10

[96] Matthias Muller, Matthias Teschner, and Markus Gross. Physically-based simulation of objects represented by surface meshes. In *CGI '04: Proceedings of the Computer Graphics International*, pages 26–33, Washington, DC, USA, 2004. IEEE Computer Society. 10

[97] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL tutorial and reference guide: C++ programming with the standard template library*. Professional Computing Series. 2nd ed. edition, 2001. 47

[98] A. Nealen, M. Muller, R. Keiser, E. Boxerman, and M. Carlson. Physically Based Deformable Models in Computer Graphics. In *Computer Graphics Forum*, volume 25, pages 809–836. Blackwell Publishing, 2006. 10

[99] A. Nealen, M. Müller, R. Keiser, E. Boxermann, and M. Carlson. Physically based deformable models in computer graphics. 2005. 10

[100] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, 2008. 41

[101] C. Nvidia. Compute Unified Device Architecture Programming Guide. *NVIDIA: Santa Clara, CA*, 2007. 83, 129

[102] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. 41

[103] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *HPCN'96*, pages 493–498, Bruxelles, 1996. Springer. 50, 68

[104] Laurent Pigeon. *Environnement Interopérable Distribué pour les Simulations Numériques avec Composants CAPE-OPEN*. PhD thesis, Institut National Polytechnique de Grenoble, 2007. 54, 56, 67

[105] Xavier Provot. Collision and self-collision handling in cloth model dedicated to design garments. *Computer Animation and Simulation*, pages 177–189, 1997. 15

[106] K. Randall. Cilk: Efficient multithreaded computing. Technical report, Cambridge, MA, USA, 1998. 52

[107] Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi. Standard templates adaptive parallel library (stapl). In David R. O'Hallaron, editor, *LCR*, volume 1511 of *Lecture Notes in Computer Science*, pages 402–409. Springer, 1998. 47

[108] J.N. Reddy. *An introduction to the finite element method*. McGraw-Hill New York, 1993. 10

[109] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007. 52

[110] James Reinders. *Intel threading building blocks : outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. 44, 46

[111] J. Revelles, Carlos Ureña, and M. Lastra. An efficient parametric algorithm for octree traversal. In *International Conference in Central Europe on Computer Graphics, Visualization and Interactive Media*, 2000. 24, 112

[112] Abdennour El Rhalibi, Steve Costa, and David England. Game engineering for a multiprocessor architecture. In *DIGRA Conf.*, 2005. 57

[113] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993. 47

[114] M. Schirski, A. Gerndt, T. van Reimersdahl, T. Kuhlen, P. Adomeit, O. Lang, S. Pischinger, and C. Bischof. Vista flowlib - framework for interactive visualization and exploration of unsteady flows in virtual environments. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, pages 77–85, New York, NY, USA, 2003. ACM. 58

[115] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *SIGGRAPH Comput. Graph.*, 20(4):151–160, 1986. 9

[116] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008. 41

[117] Satoru Shingu, Hiroshi Takahara, Hiromitsu Fuchigami, Masayuki Yamada, Yoshinori Tsuda, Wataru Ohfuchi, Yuji Sasaki, Kazuo Kobayashi, Takashi Hagiwara, Shin ichi Habata, Mitsuo Yokokawa, Hiroyuki Itoh, and Kiyoshi Otsuka. A 26.58 tflops global atmospheric simulation with the spectral transform method on the earth simulator. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–19, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 60

[118] Thomas S. Sørensen and Jesper Mosegaard. An introduction to gpu accelerated surgical simulation. In Matthias Harders and Gabor Szekely, editors, *Third International Symposium, ISBMS 2006*, volume 4072 of *Lecture Notes in Computer Science*, pages 93–104. Springer Berlin / Heidelberg, 2006. 42

[119] V. S. Sunderam. Pvm: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, 1990. 43

[120] Eduardo Tejada and Thomas Ertl. Large steps in gpu-based deformable bodies simulation. *Simulation Modelling Practice and Theory*, 13(8):703–715, 2005. 60, 83

[121] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. *Proceedings of Vision, Modeling, Visualization (VMV 2003)*, pages 47–54, 2003. 23

[122] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, Laks Raghupathi, A. Fuhrmann, Marie-Paule Cani, François Faure, N. Magnetat-Thalmann, W. Strasser, and P. Volino. Collision detection for deformable objects. *Computer Graphics Forum*, 24(1):61–81, March 2005. 12

[123] Bernhard Thomaszewski, Simon Pabst, and Wolfgang Blochinger. Parallel techniques for physically based simulation on multi-core processor architectures. *Computers & Graphics*, 32(1):25–40, February 2008. 60

[124] Henrik Tramberend. Avocado: A distributed virtual reality framework. *Virtual Reality Conference, IEEE*, 0:14, 1999. 58

[125] Gino van den Bergen. Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools*, 2(4):1–13, 1997. 13

[126] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. *Computer Graphics Forum*, 20(3):260–267, 2001. ISSN 1067-7055. 9

[127] Pascal Volino and Nadia Magnenat Thalmann. Implementing fast cloth simulation with collision response. *Computer Graphics International Conference*, 0:257, 2000. 12

[128] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, New York, NY, USA, 1992. ACM. 44

[129] Lukasz Wesolowski. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, Dept. of Computer Science, University of Illinois, 2008. http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml. 51

[130] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM. 41

[131] Terry L. Wilmarth. *POSE: Scalable General-purpose Parallel Discrete Event Simulation*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005. 59

[132] Tao Yang and Apostolos Gerasoulis. A fast static scheduling algorithm for dags on an unbounded number of processors. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 633–642, New York, NY, USA, 1991. ACM. 50, 55

[133] Thomas Y. Yeh, Petros Faloutsos, and Glenn Reinman. Enabling real-time physics simulation in future interactive entertainment. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 71–81, New York, NY, USA, 2006. ACM. 58, 76

[134] F. Zara, F. Faure, and J-M. Vincent. Physical cloth simulation on a pc cluster. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 105–112, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. 60

[135] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. Renderants: interactive reyes rendering on gpus. *ACM Trans. Graph.*, 28(5):1–11, 2009. 52