# Synchronization-Free Parallel Collision Detection Pipeline

Quentin Avril, Valérie Gouranton, Bruno Arnaldi

# Synchronization-Free Parallel Collision Detection Pipeline

Quentin Avril* Valérie Gouranton† Bruno Arnaldi‡

Université Européenne de Bretagne, France
INSA, INRIA, IRISA, UMR 6074, F-35043 RENNES

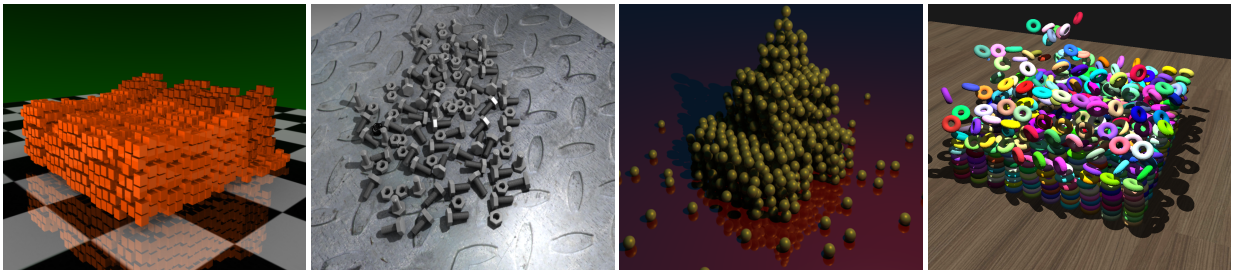

Figure 1: A sample of environments used to perform tests and to compare algorithmic performances.

**ABSTRACT**

We present a first parallel and adaptive collision detection pipeline running on a multi-core architecture. This pipeline integrates a first global synchronization-free parallelization of its major steps and enables to dynamically adapt the parallelism repartition during the simulation. We propose to break the sequentiality of the pipeline by simultaneously executing the two main phases (broad and narrow). We introduce and use a new buffer structure to share objects pairs between threads. To fully exploit multi-core performance, we propose a new dynamic load balancing technique to distribute threads among phases of the pipeline. This dynamic threads balancing acts on the broad and narrow phases in relation to their computation time. This technique favors the longest phase by giving it more CPU threads to run in parallel. Results show that this new generation of parallel pipeline enables to adapt computations to the simulation scenario evolution and to the run-time architecture. We tested our solution on a 8*cores architecture and performance measurements show that this first parallel pipeline is well-suited for the collision detection problem and enables to significantly reduce computation time compared to the sequential one.

**Index Terms:** I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling

## 1 INTRODUCTION

Collision detection is a large research field in charge of determining if two or several objects collide in a virtual environment. Collisional computations are used in several fields including computer animation, robotics, physical simulations (medical, cars industry, civil engineering...), video games and haptic applications. Virtual environments and 3D objects are constantly evolving to become increasingly large and complex. The performance level for a real time use becomes harder to ensure in large-scale virtual environments. With the recent impressive advances in hardware, algorithms for collision detection have greatly improved but remain mostly unprepared to new kind of architectures (multi-GPU, multi-processor, multi-core...).

During several years, hardware specialists have been able to increase CPU clock frequency and provide parallelism improvement in instruction sets. Thus, mono-thread applications became faster on a new generation of processors without any code modification. To have better management of power consumption and to still increase performances, they now promote multi-core architectures. Software, libraries and any piece of code must be written and specially adapted to take advantage of this hardware evolution. It is no longer possible to rely on the evolution of processing power to overcome the problem of real-time collision detection. The most efficient and fastest collision detection algorithm will not necessarily insure the best performance throughout a parallel simulation.

**Main Results:** We propose a first adaptive and parallel collision detection pipeline in which the main phases (broad and narrow) are simultaneously executed. These phases are executed in parallel and share their data through a common buffer respecting a producer-consumer pattern. This parallel collision pipeline runs on multi-core architectures and enables to dynamically balance threads among phases using a new dynamic parallelism balancing. This technique enables to adapt the parallel pipeline on a $X$-core architecture and to provide better performances.

The rest of our paper is organized as follows: In Section 2 we report related work on collision detection and parallel algorithms. In Section 3 we detail our algorithmic choices concerning the parallel pipeline. We present our parallel collision detection pipeline in Section 4. Our new technique of parallelism balancing is presented in section 5. Presentation and discussion of performance measurements are shown in Section 6. Then, we conclude and open on future work in Section 7.

*e-mail: quentin.avril@irisa.fr
†e-mail: valerie.gouranton@irisa.fr
‡e-mail: bruno.arnaldi@irisa.fr

## 2 Related Work

Collision detection problem has been intensively studied for many years in the VR field [1, 17, 27, 32]. Our review will focus on the the collision detection pipeline and parallel algorithms that have been proposed.

### 2.1 Collision Detection Pipeline

Consider $n$ moving objects in a virtual environment, to test all objects pairs tend to perform $n^2$ pairwise checks. When $n$ is large it becomes an important computational bottleneck. Collision detection can be compared as a pipeline of successive filters [16]. These filters provide an increasing efficiency and robustness all along the pipeline traversal. This concept is similar to the concept of a rendering or visualization pipeline. The input of the collision detection pipeline is a set of objects, while the output is a set of pairs of objects (and possibly polygons). This set is then transmitted to the physical response module. The pipeline is composed by two main parts: broad and narrow phase.

We present, in the following, the two phases of the collision detection pipeline and main sequential algorithms that have been proposed.

#### 2.1.1 Broad phase

The first part of the pipeline is in charge of a quick and efficient removal of objects pairs that are not in collision. Brute force approach is based on the comparison of the overall bounding volumes of objects to determine if they collide or not. In the bounding volume family many models have been proposed such as spheres [16], Axis-Aligned-Bounding-Box (AABB) [6], Oriented-Bounding-Box (OBB) [10], discrete oriented polytopes (k-DOP) [21] and many others. Other methods have been proposed to use spatial partitioning and divide space into unit cells: regular grid [29], octree [4], Binary Space Partitioning (BSP) [28] or k-d tree structure [5]. Methods that take care of the objects movement are called kinematic methods. If two objects move in opposite directions, they can not meet together [34]. Topological approach is based on the positions of objects in relation to others. One of the most used is called *"Sweep and Prune"* [7] and consists, first, in projecting objects coordinates on the environment axis (x, y, z) and then checking for overlaps between objects coordinates. Pairs with simultaneous overlapping on the three axis are transmitted to the narrow phase to check the presence or absence of collision.

#### 2.1.2 Narrow phase

The second part of the pipeline is in charge of performing an exact collision detection test between objects pairs previously filtered by the broad phase. We can classify narrow phase algorithms in four main families [23]. Algorithms working with objects primitive (faces, edges and vertices) are called feature-based algorithms. The first one appeared in 1991 [26]. It proposed to divide space around objects in Voronoi regions and to detect closest features pairs between polyhedrons. Another well-known family is simplex-based algorithms, the GJK algorithm [9] is one of the most famous that uses Minkowski difference between polyhedrons. Two convex objects collide if and only if their Minkowski difference contains the origin. Algorithms based on bounding volume are very often used in physical simulation because they provide good efficiency, quickness and they highly improve performances. Bounding volume hierarchies (BVH) enable to arrange bounding volume into tree hierarchies (binary tree, quad tree...) in order to reduce the number of tests to perform. Deformable objects are one the most important challenge for BVH because hierarchy structures have to be updated during the simulation [6, 32]. Finally, image space-based algorithms work using image-space occlusions queries that are suitable to be used on graphics hardware. They rasterize objects to perform either 2D or 2.5D overlap test in screen space [3].

### 2.2 Parallel Collision Detection

The parallel solution of collision detection algorithms is a recent field in high performance computing. In the parallel collision detection field we can distinguish two different families of algorithms, namely GPU-based and CPU-based. The pipeline has never been parallelized but Zachmann [36] made an evaluation of the performance of a theoretical parallelized back-end of the pipeline and showed that if the environment density is large compared to the number of processors, then good speed-ups can be noticed.

A solution using image-space visibility queries has been proposed for the broad phase [13]. The brute force of *"Sweep and Prune"* has also been adapted to a multi-core architecture and reduces time by 6 on a 8-core architecture [2]. Parallelization is made on AABB update coupled with the broad phase algorithm with a sequential synchronization point between both phases.

The GPU-based family is used to perform collision detection for few years using typical GPU solutions [8] but it becomes more and more used to perform non-common GPU solutions. Image-based algorithms have been proposed to exploit the growing computational power of graphics hardware. Cinder [22] is an algorithm exploiting GPU to implement a ray-casting method to detect collision. GPU-based algorithms for self-collision and cloth animation have also been introduced by Govindaraju et al. [11, 12]. An article describes the use of several GPUs during collision detection process where only one GPU is in charge of the physical pipeline, the other ones are in charge of the rendering operations [18]. Recent works use thread and data parallelism on a single GPU to perform fast hierarchy construction, updating, and traversal using tight-fitting bounding volumes such as oriented bounding boxes (OBB) and rectangular swept spheres (RSS) [24]. Algorithms using *Layered Depth Images* (LDI) to detect collision and create physical reaction have also been proposed [14, 8].

Multi-processor machines are also used to perform collision detection [20]. Depth-first traversal of bounding volumes tree traversal (BVTT) and parallel cloth simulation [30] are good instances of this kind of work. Few papers also presented multi-threading use on single processor (Lewis et al. [25] propose a multi-threaded algorithm to simulate planetary rings). Broad phase has also been developed on a Field-Programmable Gate Array (FPGA) [35].

Few papers appeared dealing with new parallel collision detection algorithms using multi-core architecture. A new task splitting approach for implicit time integration and collision handling on a multi-core architecture has been proposed [33]. Tang et al. [31] propose to use a hierarchical representation to accelerate collision detection queries and an incremental algorithm exploiting temporal coherence. The overall is distributed among multiple cores. They obtained a 4X-6X speed-up on a 8-core processor based on several deformable models. Kim et al [19] propose to use a feature-based bounding volume hierarchy (BVH) to improve performances of continuous collision detection. They also propose novel task decomposition methods for their BVH-based collision detection and dynamic task assignment methods. They obtained a 7X-8X speed-up using a 8-core architecture compared to a single-core. Hermann et al. [15] propose a parallelization of interactive physical simulations. They obtain a 14X-16X speed-up

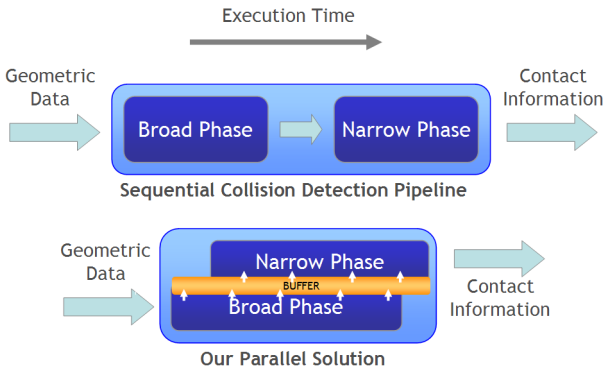


Figure 2: Overview of differences between the sequential collision detection pipeline (on top) and our parallel one (at the bottom).

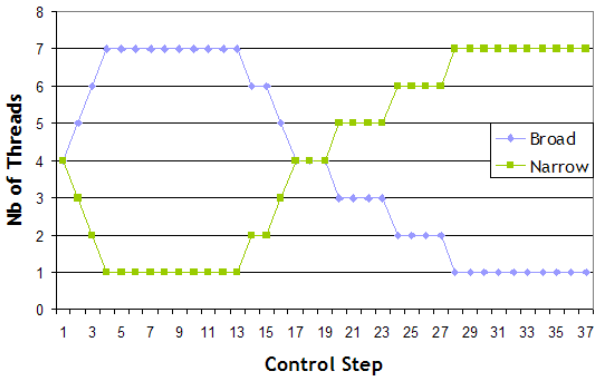


Figure 3: Example of our dynamic threads repartition between broad and narrow phase. Both phases start with 4 threads each other and we then adapt the repartition in relation to their computation time. Objects are falling to the ground at the beginning without any contact during the fall.

on a 16-core architecture compared to a single-core.

### 2.3 Positioning

Related work lets appear that many studies have been made to improve efficiency and performance of collision detection algorithms. The use of parallelism is becoming commonplace to address the problem of real-time collision detection. Thus, only fine-grain parallelizations have been done on algorithms and, for the moment, there is no work on a global parallelization of the pipeline stages and on its adaptation on any number of cores.

## 3 Technical Overview

Our parallel pipeline is technically composed by two phases: broad and narrow phase. The broad phase is based on the well-known *"Sweep and Prune"* [7] and the narrow phase has several algorithms to detect collisions between objects.

### 3.1 Broad Phase

In order to massively parallelize the broad phase algorithm, we work with the brute force method of the *"Sweep and Prune"*. This method does not update an internal structure but starts from scratch at each step. It consists in projecting objects coordinates on the environment axis (x, y, z) and checking for overlaps between objects coordinates. After projection on axis there are $\frac{(n^2-n)}{2}$ objects to test. This kind of broad phase algorithm is well-suited to the parallelization because there is no dependency between computations. They can be distributed among 2, 4, 8 or more cores without disturbing results.

### 3.2 Narrow Phase

As all objects in a simulation can be different in terms of geometric properties, our narrow phase is a kind of dispatcher in charge of finding and applying the most suitable algorithm to a pair of objects. The selection is made by taking into account geometric properties of objects like simple(cubes, spheres ...) or complex, convex or non-convex, static or dynamic, etc. . For a cube-cube detection we use a simple box-box algorithm as for spheres. For more complex convex objects, we use the algorithm of the *GJK* [9] and for non-convex objects we use a features-based algorithm based on an AABB hierarchy traversal. This dispatcher works in an easy way by taking the overall objects pairs previously filtered by the broad phase and applying the most appropriate algorithm. Its execution can also be massively parallelized by splitting and distributing the object pairs array among different cores.

Therefore, we use a pipeline equipped with highly parallelizable algorithms in order to take full advantage of multi-core architecture.

## 4 Parallel Pipeline

We present in this section our novel parallel collision detection pipeline. The execution of the two phases is not sequential but parallel. Main problems of breaking sequentiality of an algorithm are, first, to insure that there is no loss of computations and then, that results are still coherent. In our case, we keep the sequentiality of the process made on one objects pair but we break the global sequential process of the overall pairs. In other words, a pair of objects is first used by the broad phase to roughly determine if there are potential collisions. If so, it is used by the narrow phase to precise contact points. The previous sequential pipeline was waiting for the end of the broad phase to start the narrow phase. We propose to start both of them at the same time.

The two phases are linked by a buffer in which the broad phase stores its processed data. The narrow phase uses them to continue the collision detection. It is a producer-consumer pattern which is an example of resources synchronization. This pattern is especially adapted in our multi-threaded context. Figure 2 presents this new parallel pipeline and illustrates differences between the sequential one. Instead of first performing the broad phase and then the narrow phase (shown on the top of Figure 2), two control threads are created and executed in parallel on two different cores. As soon as a pair has been processed by the broad phase and considered as in potential collision, it is stored in the buffer in anticipation of being used by the narrow phase. The narrow thread begins at the end of processing the first pair of the broad phase. The shared buffer is an object pairs array where the broad phase is only allowed to put its processed pairs and the narrow phase to collect them to pursue computations. On an architecture equipped with a larger number of cores, the two father threads (broad and narrow) will be in charge of the creation and the management of several child threads.

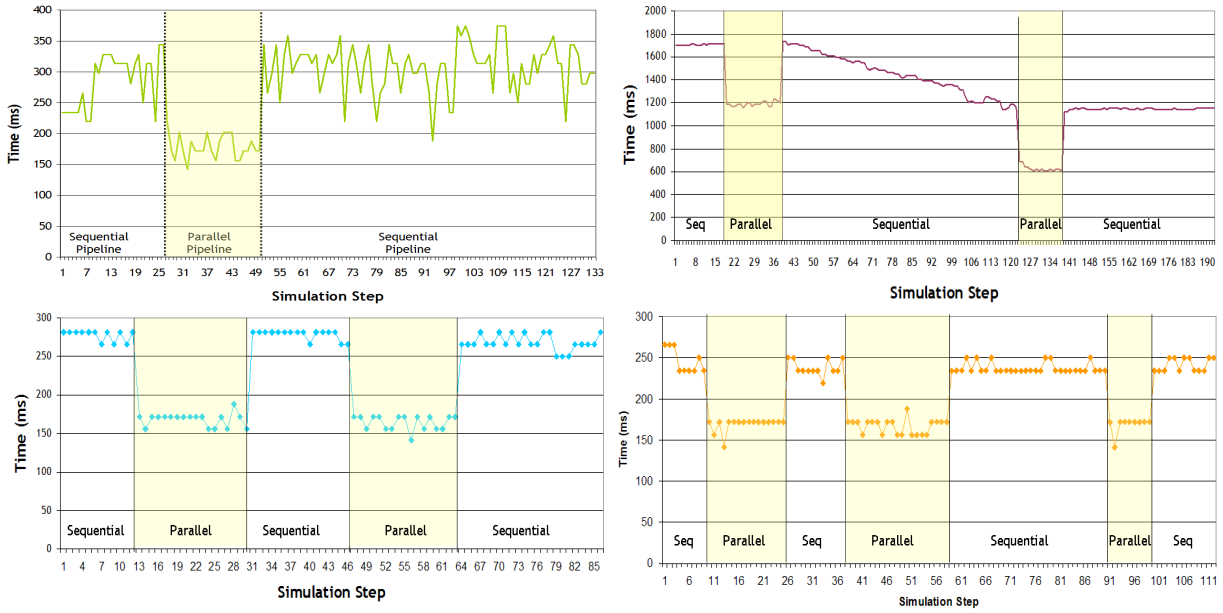


Figure 4: Four examples to illustrate difference between sequential and parallel pipeline switched during a simulation. From top left to bottom right: **a:** 500 screws and bolts in a bowl, **b:** 2000 Torus falling on to a cones environment, **c:** 3000 Cubes on a plane, **d:** 2500 Spheres on a plane.

## 5 Parallelism Balancing

The new parallel collision detection pipeline previously presented is a well-suited model for a two-core architecture. One thread in charge of the broad phase and the other one of the narrow phase. Both of them running on one core. The question is now to determine how it is possible to distribute threads on a 4, 8 or $X$-core architecture. Do we have to homogeneously separate threads or is it better to use more thread for one phase than the other one? We propose to use a new balancing method called "Parallelism Balancing". This method is in charge of determining which phase is the longest one so which one needs more thread to reduce computation time.

### 5.1 Computation Time Examples

We would naively say that narrow phase is the most time consuming phase but that is not completely true. We present two opposite examples in order to precise our purpose. The first one is a simulation with million of objects falling on to the ground. During the fall, no object is in contact with others and they are widely separated. In this case, broad phase has to check all objects at each simulation step whereas narrow phase has almost no work to do. The other example is a simulation with few complex objects that are constantly colliding each other. In this case, as there are only few objects, time spent by the broad phase is negligible whereas narrow phase time is very long due to the objects complexity. In these two opposite examples we illustrate that computation time can be different between both phases during a simulation. These two scenarios can also be found in a single simulation and we see that a dynamic adaptation would be needed. Examples also illustrate that it is hard to separate threads among phases before knowing what kind of simulation scenario we have and how the simulation will evolve.

### 5.2 Dynamic Control Step

We propose to use a dynamic parallelism balancing that adapts threads repartition during the simulation. This repartition is made thanks to time measurements achieved by a time step varying during the simulation. Analyzing time spent by phases at each simulation step is very costly, so we propose to fix a short control step at the beginning of the simulation and to extend it all along the simulation. At the beginning, the short time step is used to quickly apprehend scenario behavior and performances of the run-time architecture. We balance threads repartition in relation to time spent by both phases. When a new repartition choice is performed, we have to insure that time spent by this new repartition is better than the previous one. To do so, after each repartition change, a time control of phases is performed and frequency of the control steps is increased. After that as configuration remains unchanged, frequency decreases.

Figure 3 illustrates an example of a dynamic thread repartition during a simulation. This test was made on a 8-core architecture. Each phase starts with 4 threads which is the half of available cores. We measure time spent by phases with 4 threads and as broad phase is longer than narrow phase, one narrow thread is transmitted to the broad phase. At the beginning, all objects are falling on to the ground without contact so narrow phase has almost no computation to perform. Around the 18th control step, there is an inversion between threads repartition. This inversion means that objects are now in contact with the ground and narrow phase has more collisional computations to perform.

With this new dynamic parallelism balancing of threads during the collision detection pipeline, we are able to provide the best possible computation time all along the simulation. Examples shown in the previous section can be differentiated and processed in different ways. If these two examples of scenario are in the same simulation, we are able to dynamically adapt the parallelism repartition to face the scenario evolution. In the first example, our

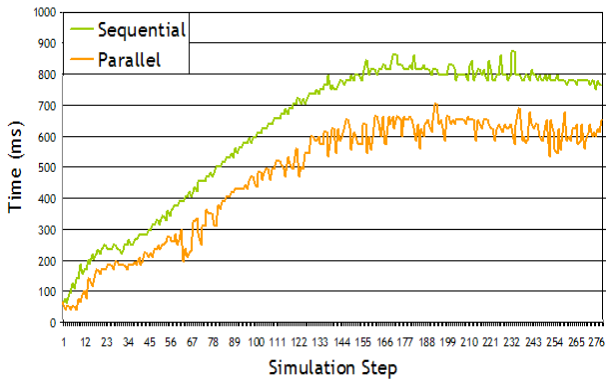


Figure 5: Comparison between time spent by the sequential collision detection pipeline and our new parallel collision one. Test was made with 1000 Torus that fall on to a plane.

balancing technique will favor a more massive parallelization of the broad phase while the second example brings a favor to the narrow phase process.

## 6 Results and Discussion

We have tested our new parallel collision detection pipeline with different simulation scenarios, going from similar objects that are completely independent to heterogeneous scenes of colliding objects (cubes, spheres, torus, screws and bolts) (Sample of our environments is shown on Figure 1). In the following, we first present results of our parallel collision detection pipeline running on a dual-core architecture, then we present results of our new method of parallelism repartition on a 8-core architecture.

### 6.1 Parallel Pipeline

In this section tests have been performed on a Intel Core 2 CPU X7900 @ 2.8GHz on Windows XP with 3GB of RAM. We compare the use of the sequential collision detection pipeline and the new parallel one. Figure 4 presents the computation time of the pipeline during simulations in which we switched between both pipelines. The simulation scenario we used is, from the top left picture to the bottom right one, 500 screws and bolts in a bowl, 2000 Torus falling on to a cones environment, 3000 Cubes falling on to a plane and 2500 Spheres falling on to a plane. These four different simulations enable to cover a wide range of geometric properties like convex or non-convex, few or several polygons, dynamic or static and simple or complex objects. For example, a torus is a non-convex object composed by 1152 polygons while a cube is a convex object composed by 12 Polygons.

The first picture on the top left of the figure 4 shows results of the simulation made with screws and bolts and we notice that the parallel pipeline enables to reduce computation time from almost 330ms to 200ms. On the right, results of the simulation made with torus illustrates a first time reduction from 1700ms to 1200ms and a second one from 1000ms to 600ms. A significant time reduction is also noticed on the two other simulations below.

A global comparison of the parallel and sequential execution of the pipeline is shown on Figure 5. We can see computation time measured at each simulation step. We notice that speed-up, brought by the parallel pipeline, is higher when sequential computation time is longer. Parallel speed-up is proportional to the sequential

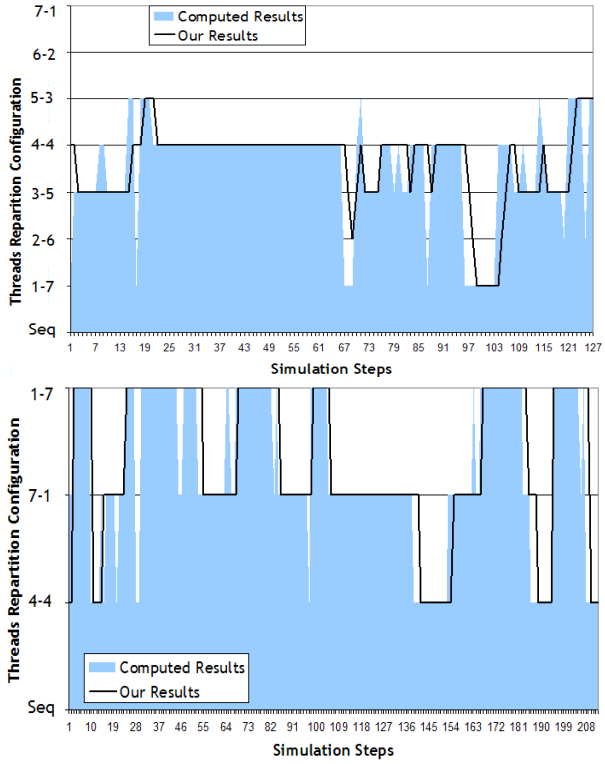


Figure 6: Results of our dynamic parallelism balancing technique that manages and adapts the threads repartition between the broad and narrow phase during a simulation. The **Top** test was made with 1000 Cubes and the **Bottom** one was performed with 500 Torus.

computation time.

We showed that in all of our test environments, the novel parallel collision detection pipeline significantly reduces computation time. On a dual-core architecture, instead of using the sequential pipeline, this new parallel pipeline will provide better performances without disturbing results.

### 6.2 Dynamic Parallelism Repartition

We present, in this section, results of our novel method of dynamic threads repartition. It is used on an architecture composed by more than two cores to fully exploit cores parallelism. Tests have been performed on a Intel Xeon CPU X5482 @ 3.20 Ghz (8-core) on Windows XP with 64GB of RAM. We first wanted to see main different between several possible configuration of threads repartition in order to justify the use of a dynamic repartition. Figure 7 is a comparison between computation time of the sequential pipeline and different configurations of threads repartition between broad and narrow phases. There are 7 different configurations to compare (1-7, 2-6, 3-5, 4-4, 5-3, 6-2 and 7-1). $X-Y$ means $X$ threads for the broad phase and $Y$ for the narrow phase.

As previously presented, the parallel solution significantly reduces computation time. It clearly appears that several configurations of threads repartition are not suitable for the type of simulation. For example, the 7-1 and 6-2 configurations in the second simulation are longer than the other ones. However, when we look for the best configuration at each time step on these curves, we notice that the best candidate is constantly changing. The

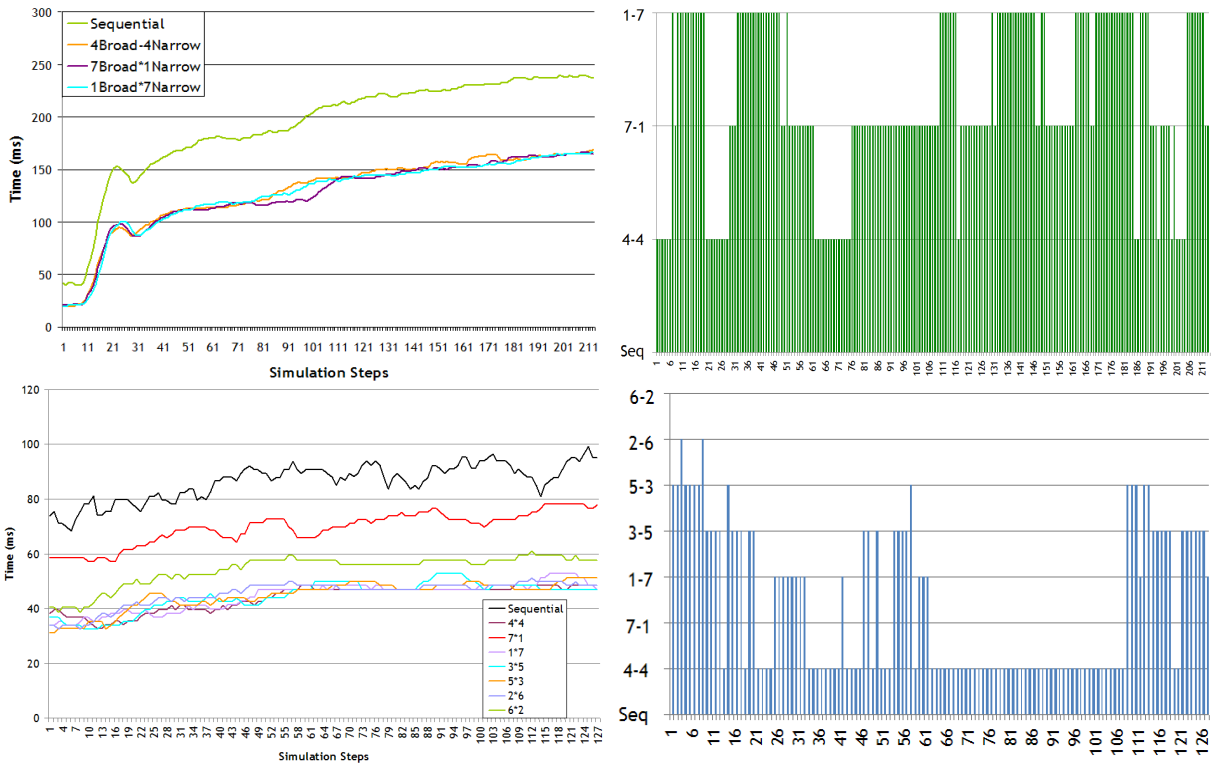


Figure 7: We compare computation time of the sequential collision detection pipeline and different configurations of our threads repartition technique (4-4 means 4 threads for the broad phase and 4 threads for the narrow phase). Tests are made on a 8-core computer with 500 Torus (**Top**) and 1000 Cubes (**Bottom**) that fall on to a plane. The **Left** pictures are computation time results and the **Right** ones are the result of the most efficient thread repartition at each step of the simulation.

two pictures on the right illustrate this phenomenon by showing at each simulation step, which threads repartition is the fastest one. We note the complete absence of the sequential execution. Depending on the evolution of the simulation, phases have got evolving requirements on the number of threads needed to reduce their computation time.

These two examples support the fact that there is not one better single threads repartition in a simulation. As broad and narrow phases have evolving requirement in terms of computation time, a dynamic adaptation is required. Figure 6 shows results that we obtained with our dynamic threads balancing technique. The black line shows the choice of threads balancing obtained by our method. To obtain the blue line we reproduced the same simulation with different thread repartition configurations and we measured computation time. We then crossed results and check which configuration obtains best results at each simulation step. Our solution closely follows real results and does not punctually change like computed results do. Limitations of our approach is also illustrated by the way that it does not anticipate time variations. For example, on the top picture, around the 67th step, we see that the adaptation takes few steps to change the threads repartition due to the dynamic control step. As explain in Section 5.2, we do not control time spent by phases at each time step. When a new configuration is chosen, the frequency of control steps becomes maximum to insure that new time is better than the previous one. And as long as configuration remains unchanged, frequency decreases. It explains the reaction time of the adaptation. In the second picture, we only gave to the repartition module 3 possibilities of threads repartition (4-4, 7-1 and 1-7). The 4-4 configuration is used when both phases spent almost the same computation time, the 7-1 configuration is to favor the broad phase and 1-7 the narrow phase.

We showed that our new parallelism balancing technique enables to adapt thread number in relation to computation time of both phases. As phases have an evolving computation time, we are able to detect it and to take it into account by adding or removing threads. The dynamic control step enables to insure that new repartition choices are correct and its varying frequency enables to spend less time controlling phases time.

## 7 Conclusion and Future Work

We have presented a first parallel and adaptive collision detection pipeline for multi-core architectures. Compared to the sequential pipeline, the two main phases of the pipeline have been developed to run simultaneously. We proposed to use a producer-consumer pattern between broad and narrow phases. Data filtered by the broad phase are stored in a buffer in anticipation of being used by the narrow phase. As soon as a pair has been processed by the broad phase, it is stored and processed by the narrow phase. This new pipeline is also coupled to a new dynamic parallelism balancing to adapt threads repartition among both phases during the simulation. An evolving control step is performed to measure time and to insure that new threads repartition is better than the previous one. This balancing technique enables to face scenario evolutions of the simulation and to reduce computation time.

We now want to test our new collision detection pipeline on larger multi-core architecture to analyze its behavior with large-

scale virtual environments. We also want to improve parallelism of the narrow phase by studying several assumptions on computations repartition.

This new parallel collision detection pipeline brings a multitude of future directions. We plan to develop a new pipeline based on this parallel one but adapted to multi-GPU architectures. It means executing phases in parallel on multi-core and multi-GPU hardware with a dynamic adaptation of tasks and data between computation units. Multi-core architectures are going to be a key component of parallel collision detection algorithms. The design of such systems requires a detailed analysis of tasks and data repartition techniques to optimize the performance of these complex run-time architectures.

## 8 Acknowledgments

The authors want to thank Florian Nouviale (INRIA Rennes) and Colin Moore (Duke University, NC) for their help in the review process and Pierre Allard (University of Rennes) for his help in graphics representation. This research is supported by INSA of Rennes (France) with Bretagne Region under project GriRV N°4295.

## References

[1] Q. Avril, V. Gouranton, and B. Arnaldi. New trends in collision detection performance. In S. R. . A. Shirai, editor, *VRIC'09 Proceedings*, pages 53–62, April 2009.

[2] Q. Avril, V. Gouranton, and B. Arnaldi. A broad phase collision detection algorithm adapted to multi-cores architectures. In S. R. . A. Shirai, editor, *VRIC'10 Proceedings*, pages 95–100, April 2010.

[3] G. Baciu and W. S.-K. Wong. Image-based collision detection for deformable cloth models. *IEEE Trans. Vis. Comput. Graph*, 10(6):649–663, 2004.

[4] S. Bandi and D. Thalmann. An adaptive spatial subdivision of the object space for fast collision detection of animated rigid bodies. *Comput. Graph. Forum*, 14(3):259–270, 1995.

[5] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACMCS*, 11(4):397–409, 1979.

[6] G. V. D. Bergen. Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools*, 2(4):1–13, 1997.

[7] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *SI3D*, pages 189–196, 218, 1995.

[8] F. Faure, S. Barbier, J. Allard, and F. Falipou. Image-based collision detection and response between arbitrary volumetric objects, Sept. 12 2008.

[9] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4:193–203, 1988.

[10] S. Gottschalk, M. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the ACM Conference on Computer Graphics*, pages 171–180, New York, Aug. 4–9 1996. ACM.

[11] N. K. Govindaraju, M. C. Lin, and D. Manocha. Fast and reliable collision detection using graphics processors. In *COMPGEOM: Annual ACM Symposium on Computational Geometry*, 2005.

[12] N. K. Govindaraju, M. C. Lin, and D. Manocha. Quick-cullide: fast inter- and intra-object collision culling using graphics hardware. *IEEE VR*, page 218, 2005.

[13] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In M. Doggett, W. Heidrich, W. Mark, and A. Schilling, editors, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 025–032, San Diego, California, 2003. Eurographics Association.

[14] B. Heidelberger, M. Teschner, and M. H. Gross. Real-time volumetric intersections of deforming objects. In T. Ertl, editor, *VMV*, pages 461–468. Aka GmbH, 2003.

[15] E. Hermann, B. Raffin, and F. Faure. Interactive physical simulation on multicore architectures. In *Eurographics Workshop on Parallel and Graphics and Visualization, EGPGV'09, March, 2009*, Munich, Allemagne, 2009.

[16] P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, Sept. 1995. ISSN 1077-2626.

[17] P. Jiménez, F. Thomas, and C. Torras. 3d collision detection: a survey. *Computers & Graphics*, 25(2):269–285, 2001.

[18] A. M. D. Jose M. Juarez-Comboni. A multi-pass multi-stage multi-gpu collision detection algorithm. In *Graphicon 2005 Proceedings*, 2005.

[19] D. Kim, J.-P. Heo, and S. eui Yoon. Pccd: Parallel continuous collision detection. Technical report, Dept. of CS, KAIST, 2008.

[20] Y. Kitamura and A. Smith. Parallel algorithms for real-time colliding face detection. *Robot and Human Communication*, pages 211–218, Nov. 07 1995.

[21] J. T. Klosowski, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, Jan. 1998.

[22] D. Knott and D. K. Pai. Cinder: Collision and interference detection in real-time using graphics hardware. In *Graphics Interface*, pages 73–80, 2003.

[23] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and R. Rowe. Collision detection: A survey. *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 4046–4051, Oct. 2007.

[24] C. Lauterbach, Q. Mo, and D. Manocha. gproximity: Hierarchical gpu-based operations for collision and distance queries. In *Computer Graphics Forum (EUROGRAPHICS Proceedings)*, volume 29, pages 419–428, June 2010.

[25] M. Lewis and B. L. Massingill. Multithreaded collision detection in java. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications (PDPTA'06)*, volume 1, pages 583–592, Las Vegas, Nevada, USA, June 2006. CSREA Press.

[26] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance calculation. Technical report, University of Berkeley, California, Mar. 19 1991.

[27] M. C. Lin and S. Gottschalk. Collision detection between geometric models: a survey. In R. Cripps, editor, *Proceedings of the 8th IMA Conference on the Mathematics of Surfaces (IMA-98)*, volume VIII of *Mathematics of Surfaces*, pages 37–56, Winchester, UK, Sept. 1998. Information Geometers.

[28] B. F. Naylor. Interactive solid geometry via partitioning trees. In *Graphics Interface '92*, pages 11–18, May 1992.

[29] Overmars. Point location in fat subdivisions. *IPL: Information Processing Letters*, 44, 1992.

[30] A. Selle, J. Su, G. Irving, and R. Fedkiw. Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction. *IEEE Trans. Vis. Comput. Graph*, 15(2):339–350, 2009.

[31] M. Tang, D. Manocha, and R. Tong. Multi-core collision detection between deformable models. In *Computers & Graphics*, 2008.

[32] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Straßer, and P. Volino. Collision detection for deformable objects. *Comput. Graph. Forum*, 24(1):61–81, 2005.

[33] B. Thomaszewski, S. Pabst, and W. Blochinger. Parallel techniques for physically based simulation on multi-core processor architectures. *Computers & Graphics*, 32(1):25–40, 2008.

[34] G. Vaněček, Jr. Back-face culling applied to collision detection of polyhedra. *The Journal of Visualization and Computer Animation*, 5(1), Jan.–Mar. 1994.

[35] M. Woulfe, J. Dingliana, and M. Manzke. Hardware accelerated broad phase collision detection for realtime simulations. *4th Workshop in Virtual Reality Interactions and Physical Simulation (VRIPHYS) (2007)*, pages 79–88, 9 Nov. 2007.

[36] G. Zachmann. Optimizing the collision detection pipeline. In *Proc. of the First International Game Technology Conference (GTEC)*, Jan. 2001.