Hochschule Reutlingen
Reutlingen University

Parallel and Distributed Computing Group
Department of Computer Science
Reutlingen University

Physically based simulation of cloth on distributed memory architectures

Bernhard Thomaszewski and Wolfgang Blochinger

(Accepted Peer-Reviewed Manuscript Version)

© 2019. This manuscript version is made available under the CC-BY-NC-ND 4.0 license
http://creativecommons.org/licenses/by-nc-nd/4.0/

The formal publication is available at:
https://doi.org/10.1016/j.parco.2007.02.008

BibTeX

```
@article{Thomaszewski2007,
      author = "Bernhard Thomaszewski and Wolfgang Blochinger",
      title = "Physically based simulation of cloth on distributed memory
               architectures",
      journal = "Parallel Computing",
      volume = "33",
      number = "6",
      pages = "377--390",
      year = "2007",
      issn = "0167-8191",
      doi = "https://doi.org/10.1016/j.parco.2007.02.008",
      url = "http://www.sciencedirect.com/science/article/pii/S0167819107000270"
}
```

Physically Based Simulation of Cloth on Distributed Memory Architectures

Bernhard Thomaszewski[a], Wolfgang Blochinger[b,*]

[a] *WSI/GRIS,*

[b] *WSI/SR*

University of Tübingen, Wilhelm-Schickard-Institute, Sand 14, D-72076 Tübingen, Germany

Abstract

Physically based simulation of cloth in virtual environments is a computationally demanding problem. It involves modeling the internal material properties of the textile (*physical modeling*) and also treating interactions with the surrounding scene (*collision handling*).

In this paper, we present an approach to parallel cloth simulation designed for distributed memory parallel architectures, particularly clusters built of commodity components. We discuss parallel techniques for the physical modeling phase as well as for the collision handling phase which are capable to significantly reduce the respective computation times.

To deal with the very fine granularity of the physical modeling phase we apply a static data decomposition approach based on graph partitioning. In order to cope with the high irregularity of the collision handling phase we employ task parallel techniques based on fully dynamic problem decomposition. We show how both techniques can be integrated into a robust parallel cloth simulation method which can deal with considerably complex scenes.

Key words: Parallel Cloth Simulation, Parallel Collision Handling, Irregular Problems, Distributed Memory Architectures

* Corresponding author.
Email addresses: `b.thomaszewski@gris.uni-tuebingen.de` (Bernhard Thomaszewski), `blochinger@informatik.uni-tuebingen.de` (Wolfgang Blochinger).

1 Introduction

In the last years, substantial research has been carried out in the field of cloth simulation. Significant improvements have been achieved on understanding the physical behavior of textiles and on deriving appropriate mathematical models along with sophisticated simulation methods. However, this progress also resulted in enormous computational demands, especially when dealing with high quality animations based on high resolution models. In this paper we investigate on employing parallel computing to meet these computational requirements.

The simulation process of cloth comprises a large number of discrete time steps. Within each such step the computation can logically be divided into two different phases:

- **Physical Modeling Phase**
  In the first phase of a simulation step internal forces resulting from deformation and external forces due to effects like gravity or wind are determined. Based on these forces, updates for nodal velocities and positions are computed according to Newton's law of motion.
- **Collision Handling Phase**
  The second phase of a simulation step is responsible for detecting and handling interactions of the garment with other objects in the scene and also interactions of the garment with itself (self-interference). Depending on the actual method used, this results in motion constraints, repulsion forces or position and/or velocity updates for individual nodes.

One difficulty of parallel cloth simulation on distributed memory architectures originates from the very fine granularity of the physical modeling phase. We employ a data-parallel method for the modeling part, especially designed for minimizing inter-processor communication. In particular, this was achieved by data decomposition techniques based on advanced graph-partitioning algorithms.

Especially with textile simulation, several immanent properties of collision handling make the parallelization of this phase most challenging. Basically, collision handling is a global problem, because any pair of processors can own interfering elements. Thus, communication cannot be limited to processors owning neighboring elements (as within the physical modeling phase). During the course of simulation, the geometry of the considered object can change significantly. This means that also communication partners are changing in a highly dynamic manner. Moreover, interaction patterns cannot be predicted and are often extremely unstructured. In this paper, we present a task parallel method for collision handling which can cope with this high degree of irregularity and which can also be tightly integrated with the physical modeling phase. To the best of our knowledge, our work represents the first research effort on parallel collision handling for textile simulation on distributed memory architectures.

The rest of our paper is organized as follows: In Section 2 we discuss related work. Section 3 gives a brief account of state-of-the-art cloth simulation methods. In Section 4 we discuss our approach to parallel cloth simulation in detail. We report on performance measurements in Section 5.

2 Related Work

2.1 Cloth Simulation

Physically based simulation is a widely adopted paradigm for reproducing the dynamic behavior of deformable surfaces like cloth. The research literature on cloth modeling is abundant and we refer the interested reader to the textbooks [1] and [2]. The seminal work of Baraff and Witkin [3] laid the ground for fast and stable cloth simulation using implicit time stepping to solve the arising ordinary differential equations. Later extensions and developments addressed further physical as well as numerical aspects [4–8]. Despite these advances, even on recent workstations the simulation of cloth with high resolution meshes (beyond 10000 vertices) is still very time consuming.

2.2 Parallel Cloth Simulation

Gutierréz *et al.* [9] report on a cloth simulation method for NUMA parallel architectures which employs an implicit integration method for the modeling phase. Lario *et al.* [10] describe a rapid parallelization approach of a multilevel cloth simulator on shared-memory architectures using OpenMP. Zara *et al.* [11] deal with parallel cloth simulation on (distributed-memory) PC clusters employing both, explicit and implicit integration techniques.

The work of Zara *et al.* is the most related to the research presented in this paper, both in terms of the employed numerical algorithms and in terms of the target parallel architecture. The other two approaches are based on shared address space parallel computers which are certainly more easy to program. However, they do not scale well and/or have a worse price/performance ratio compared to distributed-memory architectures, like clusters built of commodity components. A difference between our work and the work of Zara *et al.*

is the way problem decomposition and task mapping for the modeling phase is carried out. While we perform a completely static approach based on data partitioning which minimizes inter-processor interaction, the work of Zara *et al.* is based on partitioning dynamically generated task dependency graphs. In contrast to our work, all other approaches to parallel cloth simulation do not explicitly address collision handling. This limits their usefulness to simple scenes.

2.3 Parallel Contact Detection

Parallel contact detection has previously been studied in the context of various applications from the engineering domain, e.g. simulation of projectile penetration [12] or simulation of foam compression [13]. The basic principle of these approaches is to identify a subset of elements which can potentially get in contact. These elements are called surface elements and typically constitute only a small fraction of the total number of elements in the simulation. In [13] a separate partitioning is used during the collision handling phase which exclusively considers surface elements. Alternatively, multi-constraint, multi-objective graph partitioning algorithms are employed to avoid expensive relocation actions between the two phases [12]. In cloth simulation, every element is a surface element and it is not possible to predict the set of elements which actually interfere. Thus, approaches exclusively based on graph-partitioning are not suitable for collision handling in parallel cloth simulation.

3 Physically Based Cloth Simulation

For the realistic simulation of complex dynamic systems such as clothes there is virtually no alternative to physically based modeling. The range of existing approaches in this area is still abundant. Methods vary from very simplified models for real-time scenarios (e.g. video games) to techniques that were designed to reproduce measured material parameters in an accurate way. These latter approaches usually have high computational demands due do the numerical models used and the resolution of the meshes required. In the following paragraphs we will briefly outline the (sequential) method which forms the basis for our parallel implementation. This involves the description of the physical model, the numerical time integration scheme and the collision handling algorithm.

3.1 Physical Model

For the physical model we rely on an approach based on continuum mechanics. The basic quantities are *strain* which is a dimensionless deformation and *stress* which is a force per area or length. The two are connected via a constitutive relation, i.e. a material law which in our case is linear. The result of this approach is a partial differential equation which has to be discretized in space using numerical methods. To this end, we use a linear finite element approach as described in [7]. This yields a system (or stiffness) matrix relating nodal displacements of the mesh to forces acting on the nodes. Because only local neighbors have influence on the force on one node, this matrix is sparse. The system is extended to account for dynamic motion according to Newton's second law, leaving the following system of coupled ODEs

$$\begin{aligned}\dot{x}(t) &= v(t)\\ \dot{v}(t) &= M^{-1}f(x(t), v(t))\ .\end{aligned}$$

To obtain the dynamic evolution of the system these equation have to be stepped forward in time. In the case of cloth simulation the system equations are inherently stiff and are thus susceptible to instabilities when using explicit integration schemes. Since the work of Baraff et al. [3] the computer graphics community has settled on using implicit integration schemes for cloth simulation. The heart of this method is the solution of a sparse LES which can be carried out in a convenient way using the conjugate gradient (cg) method [14].

3.2 Collision Handling

Besides the simulation of the intrinsic properties of cloth the interaction with its environment has to be modeled. This involves the detection of any collisions and an adequate response to prevent the clothes from intersecting. The proper treatment of these two components (to which we refer as collision handling in the remainder) is a very complex task [15]. While the physical simulation engine computes new states at distinct intervals only, collisions can occur at any instant in between such intervals. Algorithms that handle these cases in a robust way are often very complex and time consuming such that the collision handling step soon becomes a bottleneck in the simulation pipeline.

Basically, detecting interference between two arbitrarily shaped objects breaks down to determining the interference between all of the primitives (i.e. faces, edges, and vertices) of one mesh with every primitive of the meshes representing the other objects. With complex objects comprising thousands of faces,

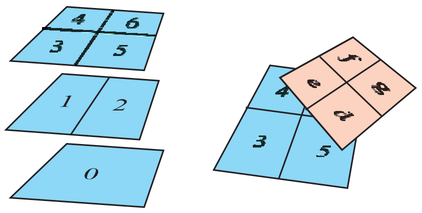


Fig. 1. Interfering objects. *Left*: Different levels of the BVH. *Right*: Overlapping Faces.

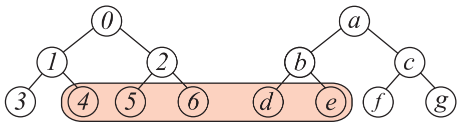


Fig. 2. BVH structure for the two objects in Fig. 1. Overlapping leaf nodes are marked.

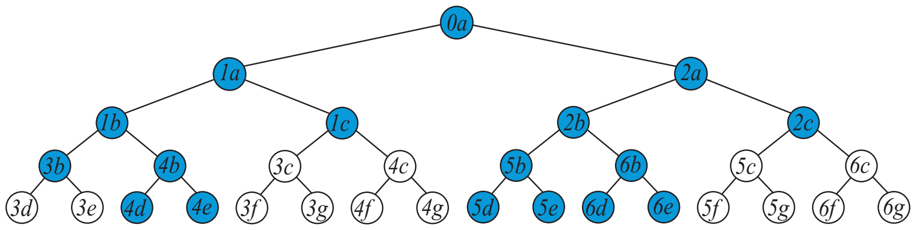


Fig. 3. Test tree for the colliding objects shown in Fig. 1.

this naïve approach soon becomes too expensive, because of its quadratic complexity wrt. the number of faces.

A common way to accelerate the interference tests is to structure the objects under consideration hierarchically with bounding volumes. Usually, a bounding volume hierarchy (BVH) is constructed for each object in the scene (including deformable as well as rigid objects) in a preprocessing step in the following way (see Fig. 1 and 2): a bounding volume enclosing the entire object is set as the root node of the tree representing the hierarchy. This node is then subdivided recursively until a leaf criterion is reached. Usually, the leaves contain one single primitive.

For our implementation we use the approach described in [16] which is based on a BVH with discrete oriented polytopes (k-Dops) as bounding volumes. More specifically, we use binary trees with 18-Dops, i.e. the bounding volumes are enclosed by 18 planes with predefined (discrete) orientation. With a BVH constructed, the test for intersection between two objects now proceeds as follows: first, the bounding volumes corresponding to the root nodes of the two hierarchies are tested for intersection. Only if these two overlap the corresponding children bounding volumes are recursively tested for intersections (see Fig. 3). Besides the interference with other objects the cloth can also intersect with itself. Basically, the same algorithms can be used to find these self collisions but here, an efficient strategy is even more important. Usually, criterions based on surface curvature are used to rule out non-intersecting parts of the cloth quickly (cf. [17]). This interference test delivers a pair of primitives that are close to each other or intersect.

A robust method to prevent the imminent intersection was presented by Bridson et al. [18]. Our own collision response is based on this approach. The essence of this algorithm is to apply a stopping impulse to approaching triangles (i.e. adjust their nodal velocities) whenever their distance falls below a certain threshold. We briefly outline our method along with some necessary extensions in the following.

The collision detection stage provides a set of close face pairs. This set is the input for the subsequent collision response phase. Every such colliding face pair is further decomposed into a set of lower level collisions among the geometric components of the faces, i.e. triangles, edges and vertices. This results in 6 vertex-triangle and 9 edge-edge collisions which are then treated separately. For each of these elementary collisions an impulse is calculated which, when applied to the involved components, prevents intersection. The direction and magnitude of the impulse is computed according to the geometric distance of the entities. A drawback of this method is that without further action multiple responses are computed for the same component. For example, an edge shared by two adjacent faces will receive impulses from both of the triangles. In the original sequential version of the algorithm this problem does not occur. Because impulses are applied immediately the resulting change in velocity influences and thus weakens subsequently computed impulses. Such an approach, however, does not translate to the parallel setting because it would lead to excessive communication. At first sight, it seems a possible alternative to accumulate the impulses for every vertex and normalize them afterwards. From our numerous experiments we learned that this approach does not lead to satisfying results. For some kinds of collisions the so generated impulses work well but for most of them (especially large planar collisions) the impulses are too weak to prevent intersection. However, a maximum projection of the impulses for every vertex yields good results in all cases. The rationale behind this is that the impulses are based on totally inelastic collisions and their magnitude is therefore always limited. Hence, the kinematic energy (and thus the magnitude of the relative velocities) of the involved vertices is never increased by applying the maximum impulse.

4 Parallel Cloth Simulation

In this section, we discuss our approach to parallel cloth simulation, focusing on the collision handling stage. We commence with a brief overview of our methodology to parallelize the physical modeling phase. Our goal is to provide

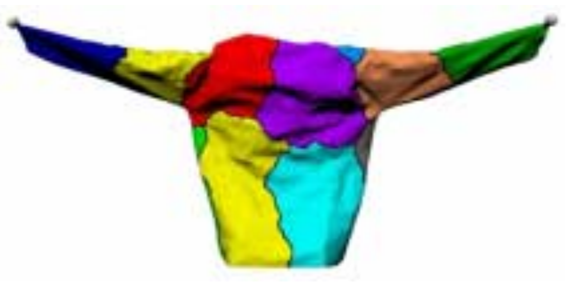


Fig. 4. Partitioning of a shirt for 12 processors.

sufficient information to illustrate the constraints which (mutually) influence the choice of parallelization strategies for each of the two phases. A complete treatment of the parallelization of this phase can be found in [19].

Subsequently, we turn to discussing the parallel techniques we have developed to speed-up collision handling. Our basic approach has been outlined in [20]. In this paper, we additionally elaborate on refined methods for problem decomposition and for load balancing. In particular, we discuss a heuristics for steering the granularity of the generated tasks and an efficient scheme for representing tasks to be executed on remote nodes. Moreover, in Section 5 we present the results of a more comprehensive experimental study. Our aim is to provide more detailed insights into the performance characteristics of our approach to parallel collision handling.

4.1 Parallel Physical Modeling

4.1.1 Problem Decomposition and Load Balancing

Compared to similar physically based simulation applications from other domains (e.g. climate modeling), problem sizes in cloth animation (in terms of the number of unknowns to be computed within each step) are typically much smaller. The resulting fine granularity represents a significant challenge for parallelizing the physical modeling phase of a cloth simulator. In order to minimize parallel run-time overhead, we apply a static problem decomposition and load balancing scheme based on data decomposition. The basic idea is to partition the vertices of the input mesh into groups of vertices with the same size and statically assign each group to one processor.

The positions of neighboring vertices which belong to different processors have to be explicitly communicated during the setup of the LES and also in every iteration of the cg procedure. Such vertices are called *ghost-points*, because they physically belong to one, but logically belong to two processors. To ensure high parallel efficiency, it is crucial to minimize communication overhead, i.e. minimizing the number of ghost-points. We employ graph partitioning

Algorithm 1 SPMD based parallel algorithm

```
partition mesh
  delivers (new) parallel numbering of vertices
redistribute initial positions vector acc. to parallel numbering
loop
  communicate ghost points
  setup LES
    compute the matrix $A$ and the right hand side vector $b$.
  solve LES
    compute the new velocities by solving $Av(t + h) = b$ with cg method
    (requires communication of ghost values)
  compute new positions
    $x(t + h) = x(t) + hv(t + h)$
  collision handling
    see Section 4.2
  if (reached frame interval) then
    all-to-one gather positions vector
    if (NODE-ID == 0) then
      permute position vector to application numbering
      generate frame
    end if
  end if
end loop
```

techniques to balance the load and at the same time to minimize communication among the processors. Figure 4 shows an exemplary partitioning of a shirt, we obtained by applying a multilevel k-way graph partitioning algorithm [21]. The partitioning process delivers a new ordering of the vertices, called the *parallel numbering*. In the parallel numbering each processor owns a consecutive range of vertex numbers resulting in a 1-dimensional, row-oriented parallel layout of the involved data structures (i.e. matrices and vectors). For generating output data, the respective data structures have to be permuted back to the application specific ordering.

After the initial decomposition stage, every processor holds a part of the global position, vertex and normal vector. The parallel simulation loop can now be executed in a synchronous SPMD (single program multiple data) fashion where every processor operates on the local parts of the data structures and the ghost points. Algorithm 1 shows the complete SPMD message passing algorithm for the physical modeling phase. It also serves us as the algorithmic framework for embedding the parallel collision handling phase (see Section 4.2).

4.1.2 Implementation

The physical modeling phase is built on top of the PETSc parallel toolkit [22,23]. PETSc is a suite of parallel data structures and routines for building scalable parallel scientific applications. It is based on the MPI (Message Passing Interface) standard and supports an SPMD (Single Program Multiple Data) style of parallel programming which is located at a higher level of abstraction than the pure SPMD message passing programming model. For mesh partitioning we use the parallel multilevel graph partitioning functionality provided by the ParMetis [21] graph partitioning library.

4.2 Parallel Collision Handling

As discussed in Section 1, the specific challenge of parallelizing the collision handling process originates from its high irregularity. Thus, extending (in a straightforward manner) the data-parallel SPMD approach of the physical modeling phase to the collision handling phase is not promising. Depending on the actual locations and types of the collisions, the amount of time spent in collision handling would differ considerably among the processors. The resulting high degree of processor idling can seriously degrade parallel efficiency of the whole execution process. Moreover, the locations of collisions can change rapidly during the course of the simulation and cannot be predicted. Consequently, static techniques for problem decomposition are not well suited. In order to deal with the outlined issues, we propose a highly dynamic, task parallel approach for accelerating the collision handling phase, which we describe in this section.

4.2.1 Parallel Collision Handling Framework

Generally, we can distinguish between two different types of collisions: external collisions and self collisions. To detect the first type we have to test our deformable object against every other (rigid or deformable) object in the scene. For the latter case, the deformable object has to be tested against itself. In this section, we show how this basic collision handling procedure can be integrated into the SPMD parallel framework of our cloth simulator.

As explained in Section 3.2, we use a bounding volume hierarchy to speed up the collision detection stage. For embedding collision handling into the SPMD framework, the BV hierarchy is built as follows: The problem decomposition stage of the physical modeling phase supplies us with a number of disjoint partitions of the vertices of the input mesh. For each processor we now proceed in the following way: a local mesh is constructed corresponding to the assigned vertices. Then, a BVH hierarchy is set up on this mesh using a top-down

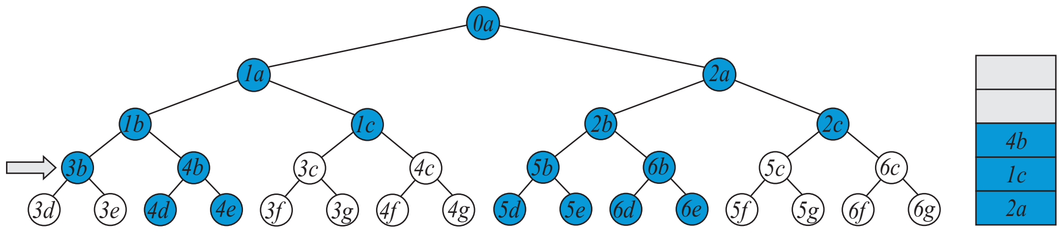


Fig. 5. Dynamic problem decomposition. The arrow indicates the current state of the BVH testing procedure. The stack on the right stores the root of untried branches.

approach. Once this is done, we combine the root nodes of the different processors to form a global hierarchy of the mesh. Testing a textile for interference with other objects is now carried out in the standard manner. First, the root node of the garment's BVH is tested against the other object's root node. If they overlap, the trees of the processors are recursively tested. This approach works well for standard collisions but for self collisions a different strategy has to be pursued. In our SPMD context we must distinguish between two different types of self collisions: namely collisions between sub-meshes of different processors and those that are real self collisions on the processor-local mesh. For the latter case we can use existing techniques since this corresponds to the usual self collision problem. For the case of inter-processor self collisions we test the corresponding BVHs against each other, similar to the way standard collisions are treated.

All discussed BVH tests form a set of *top-level tasks* of our parallel collision handling method. However, for scenes where collisions are not uniformly distributed, the number of top-level tasks and also the amount of time to execute individual top-level tasks can differ considerably among the processors. (In Section 5.2 we present examples where such uneven processor load can be observed.) In the following, we discuss a task-parallel approach which dynamically decomposes top-level tasks into smaller sub-tasks for evenly distributing the load among the processors.

4.2.2 Dynamic Problem Decomposition

Our method for dynamic problem decomposition is based on modifying the BVH testing procedure (see Section 3.2). We create sub-tasks which are responsible for detecting a subset of the collisions of the original task. Thus, the result of a sub-task is an appropriate partial collision response (in the form of impulses for the involved geometric entities). All partial responses are combined at the end of the collision handling phase by one all-to-all broadcast operation and applied to the positions vector.

For dynamically generating new tasks, every processor maintains a stack data structure which records untried branches of the BVH testing tree of the current top-level task. In the BVH testing procedure, expansion of a tree node results in two additional tree nodes each representing a refined BVH test. As in the sequential procedure, the first test is carried out by starting a new recursion level. However, before entering the recursion, the second test is pushed onto the stack. Figure 5 shows a snapshot of a BVH testing process along with the corresponding state of the stack. (In Section 4.2.3 we discuss an appropriate representation of individual BVH tests, which can be used for efficiently storing them in a data structure.)

Tests which are recorded on the stack can be executed in one of the following ways:

- A test can be removed from the top of the stack and executed sequentially when the recursion gets back to the current level. Conceptually, this case is very similar to the procedure of the original algorithm.
- One or more tests can be removed from the bottom of the stack and executed by a newly generated (sub-)task. For each assigned test, this task executes a BVH testing procedure for which the considered test defines the root of a BVH testing tree.

In oder to prevent that tasks of too fine a granularity are generated, several tests can be removed at once from the stack and assigned to a single task.

In our approach, problem decomposition is steered by the load balancing process (see Section 4.2.4). Before discussing specific details of load balancing, we first turn to the problem of finding a representation of tasks which enables efficient task transfers.

4.2.3 Task Representation

For dynamic load balancing tasks must be transferred between processors at run-time. In distributed memory architectures, task transfers require explicit communication operations. Depending on the information associated with a task, task transfers can significantly contribute to the overall parallel overhead, especially when the computation to communication ratio of the tasks is poor. Finding a compact description of tasks is therefore crucial to minimize communication overhead. In our case, the cost for transferring a task is largely determined by the representation of the associated BVH tests.

Including complete BVH information for each individual test (i.e. the entire representations of two the sub-hierarchies defined by the test) into a task would lead to a considerable task size and thus is unfeasible.

Consequently, we replicate on each node the BV hierarchy for every object in the scene such that on all nodes the identical context is provided for executing BVH tests. Particularly, all objects and all DOPS are enumerated in the same order on all processors. This enables us to represent an individual test simply as an array of integers of the form $(obj1, dop1, obj2, dop2)$. The two BV hierarchies to be tested are identified by $obj1$ and $obj2$, and the root of the test is specified by $dop1$ and $dop2$.

While this approach imposes additional costs for building the copies of the BV hierarchies at the initialization phase of the computation, the overhead during the course of the simulation is insignificant: Updating hierarchies (at the beginning of every collision handling phase) requires one all-to-all broadcast operation to provide all processors the complete positions vector. (Note that the structure of the BV hierarchy is kept fix during the whole computation.)

4.2.4 Dynamic Load Balancing

In our application, the dynamic load balancing process is responsible for triggering and coordinating task creation and task transfer operations in order to prevent that processors run idle. Basically, load balancing can employ a central controller or can be organized in a distributed manner. A central controller can establish a more accurate view of the current load of the processors, but also soon becomes the sequential bottleneck of a parallel computation. In order to achieve high scalability, we employ a fully distributed scheme. Specifically, our method is based on the distributed task pool model, i.e. every processor maintains a local task pool. Upon creation, a task is first placed in the local task pool. Subsequently, it can be instantiated and executed locally, when the processor gets idle. It also might be transfered to a remote task pool for load balancing purposes. As discussed subsequently, task creation and task transfer operations are initiated autonomously by the processors.

Task Creation Basically, tasks are generated dynamically employing the decomposition techniques discussed in Section 4.2.2. To achieve a high degree of efficiency an (active) processor must be able to satisfy task requests from other processors immediately. Additionally, we must ensure that tasks with sufficient granularity are generated. In our case, this means that when a task is to be created, the stack must contain a minimum number of BVH tests which can be assigned to the task. In order to meet both requirements, we generate tasks in a proactive fashion, i.e. independently of incoming tasks requests. Tasks are generated (and placed in the local task pool) when the size of the stack exceeds a threshold value τ. Since task generation generally imposes an overhead (even when the new task is subsequently executed on the same processor) we increase τ linearly with the current size of the task pool σ: $\tau = a\sigma + b$. This simple heuristics enables us on the one hand to provide in a timely fashion tasks with a minimum granularity (determined by b) and ensures on the other hand that the overhead of additional task decomposition operations is compensated by an increased granularity of the resulting tasks. The parameter values a and b largely depend on the parallel architecture used and should be determined experimentally. In our performance tests (see Section 5), choosing $a = 2$ and $b = 8$ delivered the best results.

A new task gets $\tau/2$ tests, where the tests are taken from the bottom of the stack. Tests are taken from the bottom of the stack for creating new tasks because such tests have a higher potential of representing a large testing tree since they originate closer to the root of the current testing tree.

Task Transfer For transferring tasks between task pools we employ a receiver initiated scheme. When a processor runs idle and the local task pool is empty, it tries to steal tasks from remote pools. First, a victim node is chosen to which a request for tasks is sent. For selecting victims we apply a round robin scheme. If available, the victim transfers a task from its pool to the local pool. Otherwise the request is rejected by sending a corresponding message. In the latter case another victim node is chosen and a new request is issued.

4.2.5 Implementation

The implementation of the previously described methods for parallel collision handling is based on the parallel system platform DOTS [24]. DOTS provides extensive support for the multithreading parallel programming model (not to be confused with the shared-memory model) which is particularly suited for task-parallel applications that employ fully dynamic problem decomposition.

DOTS Programming Model The key concept of the DOTS programming model are *thread group* objects which serve as links between different primitives of the API. Upon creation, a thread is either *explicitly* or *implicitly* placed into a thread group, calling dots_fork or dots_hyperfork, respectively. In the former case, the thread group object has to be supplied as argument to dots_fork. In the latter case, the thread is placed implicitly in the same thread group as its parent thread (which might have been created by a different processor). In both cases, a procedure to be executed by the child thread and an argument-object has to be supplied. For all subsequently applied primitives it is not relevant whether a thread has been placed explicitly or implicitly into a thread group. Threads return result objects employing dots_return. The dots_join primitive is used to retrieve results of threads from a given thread group applying *join-any* semantics: The first result which becomes available from any thread in the group is delivered. If no results are available, the calling thread is blocked until a thread of the group delivers a result. If a thread has finished execution and all result objects of the thread have been joined, it is removed from the thread group. By checking the return value of dots_join, termination of a computation can be determined.

Parallel Collision Handling Using DOTS For initiating the task-parallel execution process, we create on every processor threads which execute the top-level tasks of our parallel collision handling method (see Section 4.2.1). Threads that execute top-level tasks are created using the dots_fork primitive. All threads are placed into the same thread group. Upon completion, each thread delivers a set of impulses which represent an appropriated collision response for the corresponding top-level collision handling task.

Tasks resulting from dynamic problem decomposition (see Section 4.2.2) are modeled by DOTS threads that are created with the dots_hyperfork primitive. This approach enables us to easily synchronize with the completion of the collision handling phase regardless how many decomposition operations took place. We simply apply dots_join operations on the thread group until termination of the execution is indicated. (Note that the actual number of generated tasks largely depends on the considered scene and cannot be statically determined.)

Using the load balancing extension framework of DOTS we integrated the task transfer scheme described in Section 4.2.4. This is accomplished by implementing appropriate event-handlers defined by the framework.

5 Performance Measurements

5.1 Test Scenarios

We evaluate the performance of our approach using two test scenarios. To demonstrate the robustness of our method we focused on problems with a high degree of irregularity.

Scene 1:
For this test scene we let a round table cloth comprising 14033 vertices drape over a sphere with roughly on third of the cloth's diameter (see Figure 9). Initially, collisions only occur in a locally restricted region in the center of the cloth. As the simulation proceeds, the distribution of the collisions in the scene becomes more even. In the last part of the sequence the formation of folds exhibiting complicated self collisions can be observed. Finally, inter-processor self collisions occur at the border regions of the cloth.

Scene 2:
This scene consists of a square piece of cloth with 14641 vertices draping over a thin ondulated bar which is posed at some distance to a floor (see Figure 10). Again, the collisions are locally restricted in the first part of the simulation. Due to the special shape of the bar complicated folding patterns are formed as the cloth falls further downwards. When the cloth reaches the floor, collision occur more widespread in the mesh and the setting becomes more regular. In this scenario rigid collisions are initially predominant. As the simulation proceeds, fairly complex self collisions are produced: the cloth folds over itself while it slides towards the troughs of the bar.

5.2 Tests and Results

We used a Linux based cluster for carrying out performance measurements. All compute nodes are equipped with Intel Xeon processors (2.667 GHz) and with 2 GB of main memory. The nodes are connected by a Myrinet-2000 network.

All program runs compute 25 frames of our test scenes, where a single frame comprises 40 simulation steps. For each investigated setting, the presented performance results are based on the arithmetic mean of the wall-clock times of three individual parallel runs. Time values given for one processor are based on the sequential version of our application. (It employs sequential data structures and sequential arithmetic operations for the physical modeling phase and performs no dynamic problem decomposition during the collision handling phase.)

Figure 6 and Figure 7 depict the results of the performance measurements for our two test scenarios. Despite the high degree of irregularity of both scenes, the overall computation time as well as the individual run-times of the physical modeling and the collision handling phases can be substantially reduced by parallel execution.

In all our test cases, the time required for collision handling is greater than the time spent in physical modeling. For both scenes, the run times of the physical modeling phases are comparable. Due to the increased occurrence of self-collisions in scene 1, times needed for collision handling (and thus the total run times) are noticeably greater than in scene 2.

Independently of the number of processors (and the number of computed frames), we observe a constant amount of time (scene1: about 140 s, scene2: about 80 s) spent in preprocessing (essentially mesh partitioning and setting up BV hierarchies). With an increased total run time (i.e. computing more frames) this sequential fraction of the computation becomes less dominant. In typical production runs (with several hundreds of frames) we can expect that

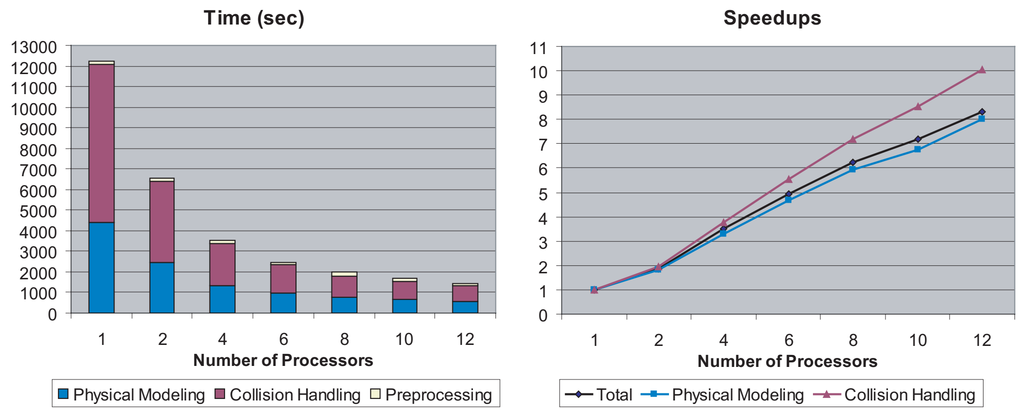


Fig. 6. Results of performance measurements for scene 1.

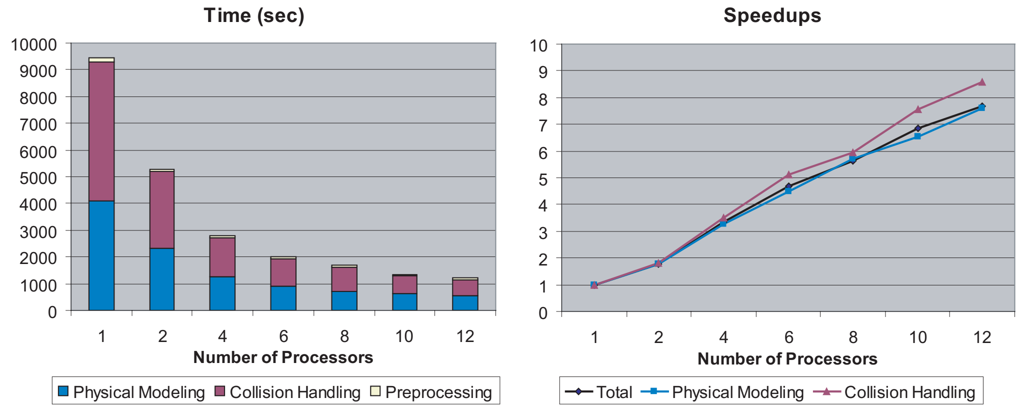
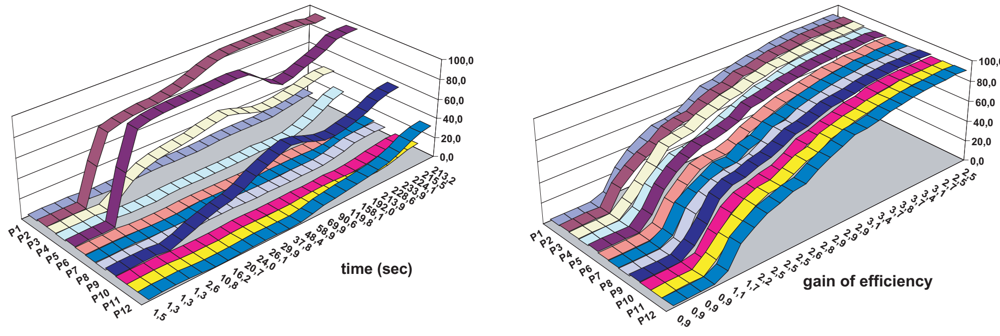


Fig. 7. Results of performance measurements for scene 2.

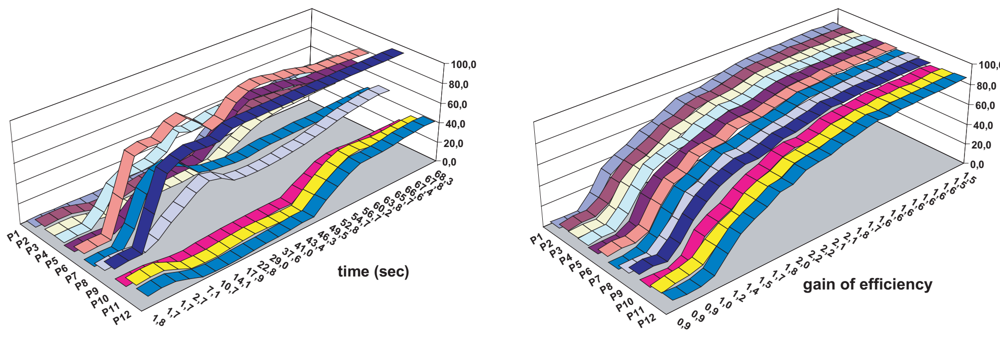
the individual speedups of the physical modeling and of the collision handling phase contribute to the overall speedup in a largely undiminished way.

In Figure 8 we compare two program runs on 12 processors for each of our two test scenes. The diagrams illustrate the average cpu utilization of each processor for the collision handling phases of all computed frames. For the program runs depicted in the diagrams on the left side, problem decomposition and load balancing were disabled. The diagrams on the right side show the program runs with dynamic problem decomposition and load balancing. Additionally, the left diagrams give the total time spent in collision handling for each frame (comprising 40 individual collision handling phases), and the right diagrams give the improvement of parallel efficiency obtained for computing the respective frame employing dynamic problem decomposition. Note that the times spent in collision handling increase by one to two orders of magnitude when complex collisions are encountered in the scene.

As the tests reveal, employing dynamic problem decomposition and load balancing for the collision handling phase can considerably improve parallel ef-



(a) Scene 1



(b) Scene 2

Fig. 8. Effect of dynamic problem decomposition and load balancing on the parallel efficiency of the collision handling phase.

ficiency, especially when intricate and unevenly distributed (self-)collisions appear in the scene. For the first few frames, where no interactions occur at all, we observe slight slowdowns which reflect the overhead introduced by our highly dynamic method.

6 Conclusion

We discussed a parallel approach to cloth simulation for distributed memory architectures, which comprises parallelization of physical modeling as well as of collision handling. The main contribution of our paper is a novel method for parallel collision handling which is capable to cope with the high irregularity exhibited with cloth simulation. In our approach problem decomposition and load balancing are tightly interrelated, realizing self-adapting parallelism. For regular scenes where collisions are evenly distributed on the processors, the overall amount of dynamic parallelism is limited. In contrast, if processors run idle due to an uneven distribution of the collisions, additional parallelism is generated and balanced over the processors. We seamlessly integrated the task parallel collision handling phase into the SPMD data parallel framework imposed by the physical modeling phase. Thus, our parallel approach to cloth simulation represents an example where considerably dissimilar types of parallelism could be beneficially combined within one application.

References

[1] P. Volino, N. Magnenat-Thalmann, Virtual Clothing, Springer, 2000.

[2] D. H. House, D. E. Breen (Eds.), Cloth Modeling and Animation, A K Peters, 2000.

[3] D. Baraff, A. Witkin, Large Steps in Cloth Simulation, in: Computer Graphics (Proc. SIGGRAPH), 1998, pp. 43–54.

[4] B. Eberhardt, O. Etzmuß, M. Hauth, Implicit-Explicit Schemes for Fast Animation with Particle Systems, in: Eurographics Computer Animation and Simulation Workshop, 2000.

[5] P. Volino, N. Magnenat-Thalmann, Comparing Efficiency of Integration Methods for Cloth Animation, in: Computer Graphics International Proceedings, 2001.

[6] K.-J. Choi, H.-S. Ko, Stable but Responsive Cloth, in: Computer Graphics (Proc. SIGGRAPH), 2002, pp. 604–611.

[7] O. Etzmuß, M. Keckeisen, W. Straßer, A Fast Finite Element Solution for Cloth Modelling, Proc. Pacific Graphics.

[8] M. Hauth, O. Etzmuß, A High Performance Solver for the Animation of Deformable Objects using Advanced Numerical Methods, in: Computer Graphics Forum, 2001, pp. 319–328.

[9] E. Gutierréz, S. Romero, L. F. Romero, O. Plata, E. L. Zapata, Parallel techniques in irregular codes: cloth simulation as case of study, Journal of Parallel and Distributed Computing 65 (4) (2005) 424–436.

[10] R. Lario, C. Garcia, M. Prieto, F. Tirado, Rapid Parallelization of a Multilevel Cloth Simulator Using OpenMP, in: Third European Workshop on OpenMP, 2001.

[11] F. Zara, F. Faure, J.-M. Vincent, Parallel simulation of large dynamic system on a pcs cluster: Application to cloth simulation, International Journal of Computers and Applications 26 (3).

[12] G. Karypis, Multi constraint mesh partitioning for contact/impact computations, in: Proc. of the 2003 ACM/IEEE Conf. on Supercomputing, IEEE Computer Society, Washington, DC, USA, 2003, p. 56.

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

[14] J. R. Shewchuck, An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, http://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.ps (1994).

[15] M. Teschner, B. Heidelberger, D. Manocha, N. Govindaraju, G. Zachmann, S. Kimmerle, J. Mezger, A. Fuhrmann, Collision Handling in Dynamic Simulation Environments, in: Eurographics Tutorials, 2005, pp. 79–185.

[16] J. Mezger, S. Kimmerle, O. Etzmuß, Hierarchical Techniques in Collision Detection for Cloth Animation, Journal of WSCG 11 (2) (2003) 322–329.

[17] P. Volino, N. Thalmann, Collision and Self-Collision Detection: Efficient and Robust Solutions for Highly Deformable Surfaces, in: Comp. Animation and Simulation, 1995.

[18] R. Bridson, R. P. Fedkiw, J. Anderson, Robust Treatment of Collisions, Contact, and Friction for Cloth Animation, in: Computer Graphics (Proc. SIGGRAPH), 2002, pp. 594–603.

[19] M. Keckeisen, W. Blochinger, Parallel implicit integration for cloth animations on distributed memory architectures, in: Proc. of Eurographics Symposium on Parallel Graphics and Visualization 2004, Grenoble, France, 2004.

[20] B. Thomaszewski, W. Blochinger, Parallel simulation of cloth on distributed memory architectures, in: Proc. of Eurographics Symposium on Parallel Graphics and Visualization 2006, Braga, Portugal, 2006.

[21] G. Karypis, V. Kumar, Parallel Multilevel k-way Partitioning Schemes for Irregular Graphs, Tech. Rep. 036, Minneapolis, MN 55454 (May 1996).

[22] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A. M. Bruaset, H. P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhauser Press, 1997, pp. 163–202.

[23] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory (2002).

[24] W. Blochinger, W. Küchlin, C. Ludwig, A. Weber, An object-oriented platform for distributed high-performance Symbolic Computation, Mathematics and Computers in Simulation 49 (1999) 161–178.

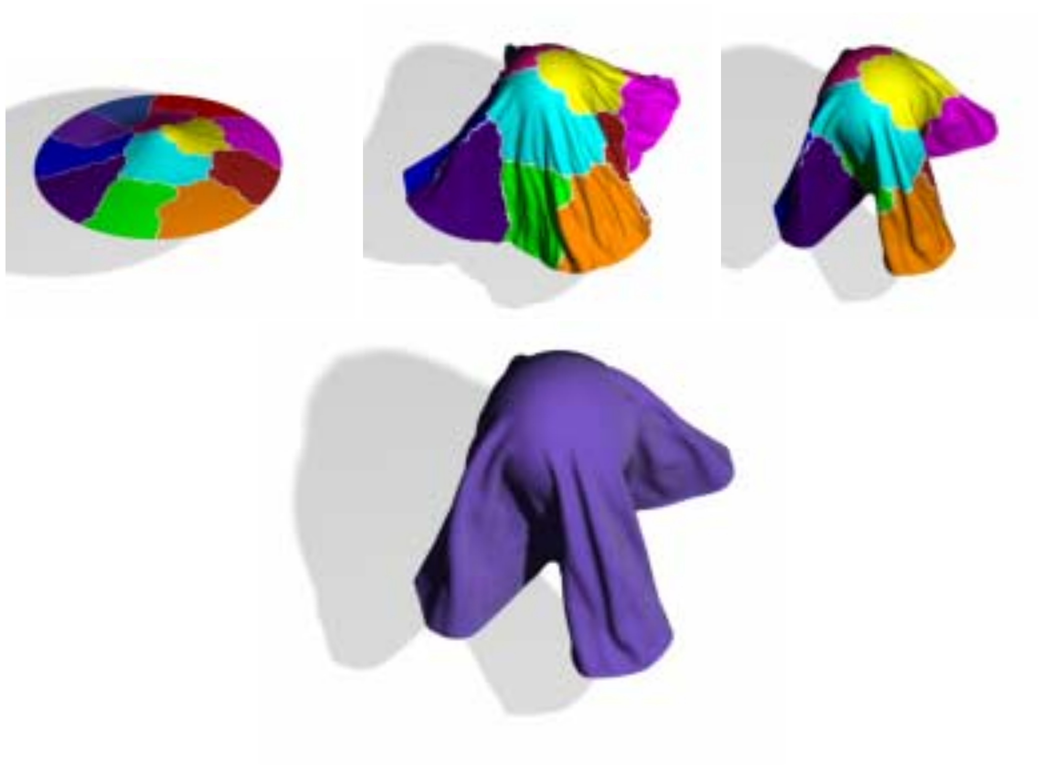


Fig. 9. Test Scene 1 (Colors in upper row indicate mesh partitioning)

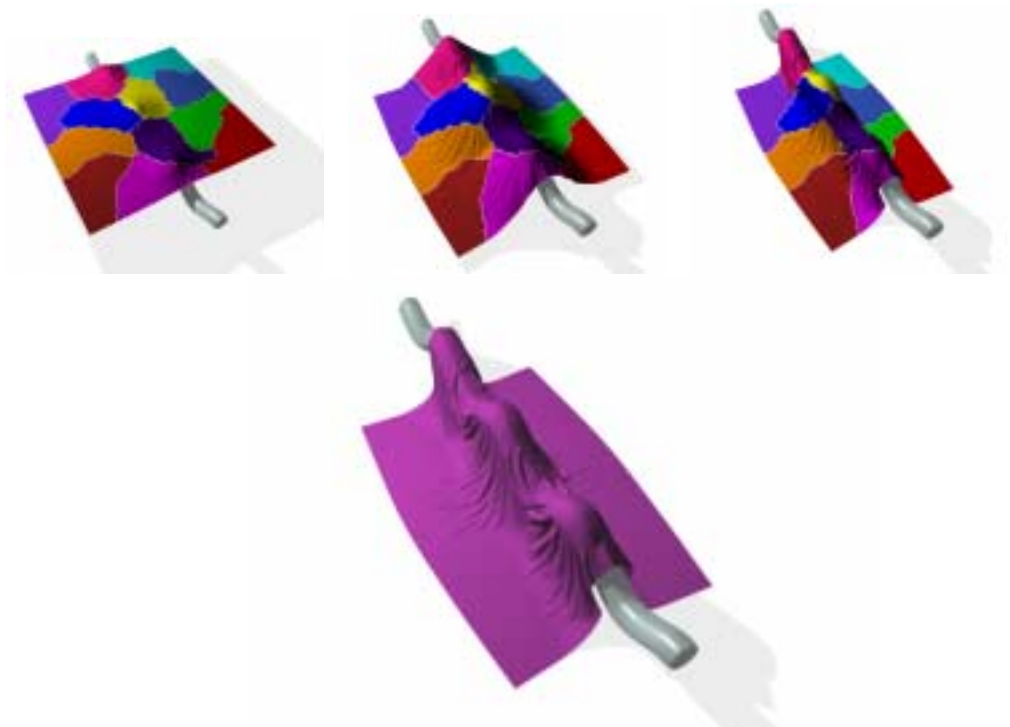


Fig. 10. Test Scene 2 (Colors in upper row indicate mesh partitioning)