Parallel Techniques for Physically-Based Simulation on Multi-Core Processor Architectures

Bernhard Thomaszewski [a,*] Simon Pabst [a] Wolfgang Blochinger [b]
[a] *WSI/GRIS, Universität Tübingen, Germany*
[b] *Symbolic Computation Group, Universität Tübingen, Germany*

Abstract

As multi-core processor systems become more and more widespread, the demand for efficient parallel algorithms also propagates into the field of computer graphics. This is especially true for physically-based simulation, which is notorious for expensive numerical methods. In this work, we explore possibilities for accelerating physically-based simulation algorithms on multi-core architectures. Two components of physically-based simulation represent a great potential for bottlenecks in parallelisation: implicit time integration and collision handling. From the parallelisation point of view these two components are substantially different. Implicit time integration can be treated efficiently using static problem decomposition. The linear system arising in this context is solved using a data-parallel preconditioned conjugate gradient algorithm. The collision handling stage, however, requires a different approach, due to its dynamic structure. This stage is handled using multi-threaded programming with fully dynamic task decomposition. In particular, we propose a new task splitting approach based on a reasonable estimation of work, which analyses previous simulation steps. Altogether, the combination of different parallelisation techniques leads to a concise and yet versatile framework for highly efficient physical simulation.

Key words:
Physically-Based Simulation, Parallel Collision Detection, Parallel Conjugate Gradients, Multi-Core Processors

1. Introduction

Physically-based simulation is an important component of many applications in current research areas of computer graphics. The most prominent examples are fluid, soft body, and cloth simulation. All of these applications utilise computationally intensive methods and runtimes for realistic scenarios are often excessive. Obviously, increasing the realism of the simulation by using more accurate methods, like finite elements, further aggravates the problem. In this paper we investigate on parallel techniques for improving the performance of physically-based simulation codes on multi-core architectures.

Generally, most of the computation time is spent on two stages, time integration and collision handling. In the following, we will therefore consider these two major bottlenecks, which are present in almost every physically-based simulation. Although we focus on cloth simulation in this work, the techniques proposed herein transfer to many other applications, like e.g. thin-shell and three-dimensional soft body simulation.

1.1. *Implicit time integration*

Often, the physical model at the centre of a specific simulator gives rise to stiff differential equations with respect to time. For stability reasons implicit schemes are widely accepted as the method of choice for numerical time integration (cf. [1]). Implicit schemes require the solution of a (non-)linear system of equations at each time step. As a result of the spatial discretisation, the matrix of this system is usually very sparse. There are essentially two alternatives for the numerical solution of the system. One is to use an iterative method such as the popular conjugate gradients (cg) algorithm [2]. Another is to use direct sparse solvers, which are usually based on fill-reducing reordering and factorisation. The cg-method is favoured in computer

* Corresponding author.
Email address: b.thomaszewski@gris.uni-tuebingen.de (Bernhard Thomaszewski).
URL: www.gris.uni-tuebingen.de/∼thomasze (Bernhard Thomaszewski).

graphics as it offers much simpler user interaction, alleviates the integration of arbitrary boundary conditions and allows balancing accuracy against speed. We will therefore focus on the cg-method in this work.

1.2. *Collision Handling*

Most practical applications for deformable objects include collision and contact situations. Maintaining an intersection-free state at every instant is of utmost importance in this context and involves the detection of proximities (collision detection) and the reaction necessary to prevent interpenetrations (collision response). In the remainder, we refer to these two components collectively as *collision handling*. We usually distinguish between external collisions (with other objects in the scene) and self-collisions. Both of these types require specifically tailored algorithms for an efficient treatment. Even with common acceleration structures (see Sec. 3.4) these algorithms are still computationally expensive. For complex scenarios with complicated self-collisions, the collision handling can easily make up more than half of the overall computation time. It is therefore a second bottleneck for the physical simulation and hence deserves special attention.

1.3. *Overview and Contributions*

In our previous work towards distributed-memory architecture we developed basic parallelization strategies for the two components of physical simulation. For the time integration stage, which exhibits a very fine granularity, we proposed a static data-parallel approach. For the highly irregular collision handling stage we proposed a dynamic task-parallel approach. In [3] the reader will find more details on these design decisions. The purpose of this work is to design efficient implementations of these state-of-the-art parallel techniques for physical simulation on shared-memory based multi-processor systems. We focus on the computationally most expensive components, which are numerical time integration and collision handling.

Implicit time integration leads to the solution of linear systems, for which we propose a parallel preconditioned conjugate gradient algorithm. We developed an efficient implementation of this method and provide a detailed explanation of preconditioner application and matrix-vector-multiplication [4]. In contrast to this, our parallel numerics code for distributed memory architectures employs the message passing based programming model provided by the PETSc toolkit [5]. Although this code could theoretically be ported to shared memory platforms, our aim here is to explicitly take advantage of the multi-core architecture, which enables a considerably different programming approach. As an example, inter-task communication can be implemented far more efficiently in the shared memory setting by simply sharing data structures among threads. Furthermore, our shared memory numerics code is entirely based on OpenMP directives. Therefore, it is much easier to integrate into existing sequential simulator code. It would require time-intensive redesign when adapting the code to the *single program multiple data*-paradigm of distributed memory architectures.

The performance of our numerical algorithms can be further increased using single precision arithmetic. We discuss this aspect in detail and explain how to implement the required modifications. Additionally, we investigate the performance of the parallel numerics code when applied to large input data.

For parallel collision handling we discuss and evaluate a novel task decomposition scheme based on temporal coherence data. In particular, we take advantage of the tight coupling of multi-core processors to derive work estimates for tasks with very low overhead. Moreover, we show how the resulting highly dynamic task-parallel execution process can be efficiently mapped to shared memory architectures by using lock-free synchronisation mechanisms for task management. We describe the implementation of this technique based on specific atomic processor instructions and experimentally assess the resulting performance gain.

Finally, we present extensive experimental studies for all of the presented methods on three recent multi-core systems, showing that our approach scales well on different platforms.

2. Related Work

Parallel Numerics. The parallel solution of large sparse linear systems is a well explored but still active field in high performance computing. Most of the work from this field focuses on problem sizes that are considerably larger than the ones dealt with in computer graphics. Therefore, standard techniques do not necessarily translate directly to our application area. In general, good overviews on parallel numerical algebra can be found in the textbook by Saad [6] and the report compiled by Demmel et al. [7]. Parallel implementation of sparse numerical kernels like the ones used in this work, has already been investigated by O'Hallaron [8]. Oliker et al. [9] explored node ordering strategies and programming paradigms for sparse matrix computations. However, they did not consider parallel preconditioning.

Parallel Cloth Simulation. Previous research on parallel cloth simulation addressed shared address-space [10–12] as well as message passing-based architectures [13–16]. Since the present article specifically deals with multi-core CPUs, we will restrict our discussion of related work to approaches designed for shared address-space machines. For a discussion of the different approaches for distributed memory architectures we refer the reader to [3].

Lario et al. [11] described the parallelisation of a cloth simulator that employs multilevel techniques. The authors focused on the time integration stage and did not address parallel collision detection. Particularly, they provided a

comparison between message passing-based and thread-based parallelisation of multilevel methods on different shared address-space architectures.

Mujahid et al. [17] addressed the parallelisation of a cloth simulation method, which is based on adaptive mesh refinement/coarsening. The resolution of the mesh is dynamically adapted so that, on the one hand, it represents the cloth with minimum computational costs and, on the other hand, the realism of the simulation is preserved. Load balancing is achieved by maintaining lists of active mesh nodes, which are equally distributed among the processors employing the dynamic work-sharing constructs of OpenMP. In contrast to our work their contribution does not deal with parallel collision handling.

The work of Gutierréz et al. [12] paid special attention to histogram reduction computations, which can be found at the core of numerical simulation codes like cloth simulation. The authors presented a framework for partitioning-based methods on NUMA machines, which exploits data affinity. In the context of this framework, several methods for parallel reduction are applied to the force computation loop of a cloth simulator and compared with each other. While their work concentrated on optimising a specific aspect, our approach encompasses all of the computation-intensive components of physically-based simulation.

Romero et al. [10] presented a parallel cloth simulator designed for non-uniform memory access (NUMA) architectures. Their work addressed the parallelisation of time integration and collision handling. While the approach taken for time integration is similar to our work, the way collision handling is carried out differs significantly. In their work, parallel collision handling is implemented by a data-parallel strategy which partitions lists of potentially colliding primitives. These lists are maintained by heuristics. Bounding volume hierarchy tests (see Sec. 3.4) are only carried out to initialise the lists and in cases where the size of the lists exceeds a given threshold. In contrast, a primary design goal of our approach is to achieve good performance and parallel efficiency for a wide spectrum of scenes, in particular for scenes with rapidly changing and challenging collision situations. As a consequence, we perform a complete series of bounding volume hierarchy tests in every collision handling phase and iterate until all collisions have been resolved. However, this strategy requires parallelising the bounding volume hierarchy testing procedure. Due to the hierarchical and irregular nature of these tests, we apply a task-parallel method which is based on fully dynamic problem decomposition.

3. Physically-based Cloth Simulation

The methods described in this article apply to any specific approach provided it uses implicit time integration and collision handling based on bounding volume hierarchies. Details on the modules of the cloth simulation system used in this work are presented below.

3.1. *Simulation Outline*

In order to provide context, we will start with a brief overview of the simulation loop. Fig. 1 shows a schematic view of the simulation loop as used in our specific implementation. The simulation starts with an initialisation stage in which meshes for deformable and non-deformable objects are loaded and bounding volume hierarchies are constructed. Each iteration of the loop begins with the assembly of the linear system of equations arising from the implicit time integration scheme. Technically, this amounts to assembling the matrix and the right hand side of the system. The basis for parallel matrix assembly is the domain partitioning described in Sec. 4.1. The actual parallel implementation simply corresponds to the sequential algorithm applied to each partition in parallel. The entries of the matrix are computed according to the underlying physical model. The next step is the solution of the system with the method of conjugate gradients, which will supply us with candidate nodal velocities. The preconditioner is first updated with the current matrix data before the iteration starts. The parallel version of this operation follows again directly from the problem decomposition. Each iteration of the cg-method involves the application of the preconditioner (*pc_apply*) as well as sparse matrix vector multiplication (*spmv*). Note that for the sake of brevity, we omit further operations, such as dot products, which need to be carried out in each iteration step. After convergence, we are provided with updated nodal velocities and compute new positions.

The updated positions are then passed on as candidate positions to the collision handling stage, where they are used to update the bounding volumes. Subsequently, the collision detection scheme is invoked and intersection-preventing responses are generated as required. Collision handling is usually invoked after numerical time integration, but it can also be directly integrated into the solution of the linear system (see [1]). This stage yields the final, collision-free nodal positions and velocities, which are used as the initial values for the next step of the simulation loop. Lastly, the final positions are also used to create output geometry for visualisation. The following subsections describe the key steps in greater detail.

3.2. *Physical Model and Kinematics*

Creating an animation amounts to computing snapshots of the geometry in the scene at discrete instants in time. We let $\mathbf{x}(t{+}h)$ stand for the discrete sampling of the nodal trajectories $x(t)$, where h is the time step and bold face letters denote discrete vectorial quantities. Point-wise kinematics can be cast as a sequence of coupled initial value problems

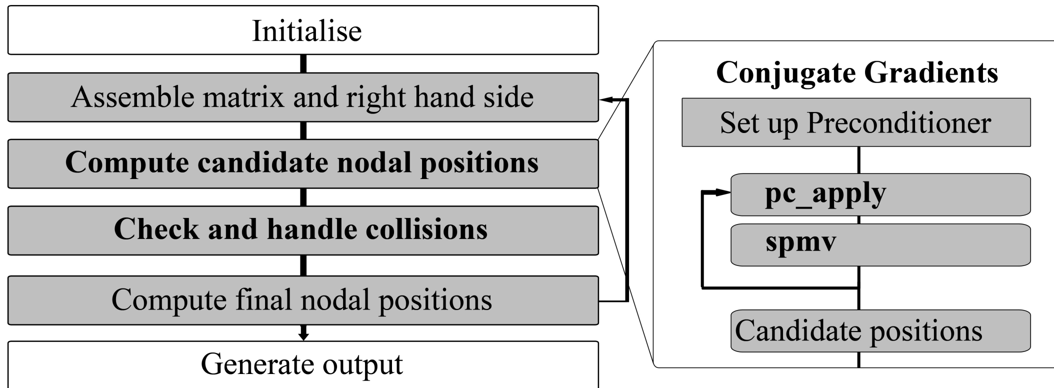

Fig. 1. Schematic view of the simulation loop according to our implementation. A grey background denotes the components that have been paralellised. The computationally most expensive parts are printed in bold face.

$$\mathbf{x}(t+h) = \mathbf{x}(t) + \int_t^{t+h} v(t)\ dt \qquad (1)$$
$$\mathbf{v}(t+h) = \mathbf{v}(t) + \int_t^{t+h} a(t)\ dt,$$

where $\mathbf{v}$ and $v(t)$ denote discrete and continuous nodal velocities, respectively. The nodal accelerations $a(t)$ are readily related to forces $f(t)$, using Newton's second law. The actual way in which these integral equations are transformed to discrete algebraic equations depends on the numerical time integration scheme. In any case, a method to compute the internal nodal forces $\mathbf{f}(t)$ at a given time t is required and we refer to it as the *physical model*.

The basis for the physical model used in our implementation is a continuum mechanics formulation of linear elasticity theory [18]. The central quantities in this case are *strain*, which is a dimensionless deformation measure, and *stress*, which is a resulting force per area. These two variables are related to each other through a material law, which in our case is simply linear. The resulting partial differential equation is discretised using a linear finite element approach as described in [19]. For dynamic simulation, inertia effects have to be included, as well as viscosity and possibly external forces.

3.3. *Numerical Time Integration*

The stiffness of the differential equations (1) suggests using implicit numerical time integration. We adopt the first order accurate implicit Euler scheme, which seeks to find $\mathbf{v}(t+h)$ and $\mathbf{x}(t+h)$ such that

$$\mathbf{v}(t+h) = \mathbf{v}(t) + h\ \mathbf{M}^{-1}(\mathbf{f}(t, \mathbf{x}(t+h), \mathbf{v}(t+h)) + \mathbf{f}_{ext}) \qquad (2)$$
$$\mathbf{x}(t+h) = \mathbf{x}(t) + h\ \mathbf{v}(t+h)\ .$$

Here, $\mathbf{M}$ denotes the diagonal mass matrix and $\mathbf{f}_{ext}$ accounts for external forces like gravity. Note that the internal forces $\mathbf{f}$ are evaluated at the end of the time step, giving rise to a system of implicit equations. Generally, $\mathbf{f}$ is a nonlinear function in terms of $\mathbf{x}$ and $\mathbf{v}$ and the system has to be solved using Newton's method. Anyhow, this breaks down to repeatedly solving linear systems. In our particular case, Eq. (2) can be cast into a formulation which is linear with respect to positions and velocities (see [19]). Hence, we need only solve one linear system per time step. The actual parallel solution of this system using the cg-method is described in Sec. 4.

3.4. *Collision Handling*

Collision Detection. As a first step, possible interferences have to be detected for the deformable objects in the scene. Since all objects are represented as polygonal meshes, this could be accomplished by testing every pair of primitives, i.e. polygons, geometrically for intersection. Because the average runtime of this naive approach is unacceptably high, bounding volume hierarchies are usually used for acceleration [20]. In this way, non-intersecting parts are quickly ruled out for a given object pair. Such hierarchies consist of two components: a tree representing the topological subdivision of the object into increasingly finer regions and bounding volumes enclosing the geometry associated with every node in the tree. In our implementation we use discrete oriented polytopes (k-DOPs) as bounding volumes (see [21,22]).
Testing two objects for interference using bounding volume hierarchies is a recursive process. First, the bounding volumes associated with the roots of the two hierarchies are tested for intersection. Only if they overlap, are the respective children tested recursively against each other. Finally, the leaves of the tree need to be checked for intersection using exact geometric tests. If a test signals close proximity or intersection, an appropriate collision response has to be generated.

Collision Response. Generally speaking, the task of the collision response stage is to prevent intersections. There are various methods for achieving this, ranging from motion constraints over repulsion forces to stopping impulses. Constraints are simple to enforce and do a good job when it comes to preventing intersections with external objects in

rather simple scenes. However, releasing constraints is usually cumbersome and often leads to nodes being arbitrarily fixed at some point in space. This is particularly disturbing for self-collisions and literally breaks the simulation.

In our implementation we therefore use a combination of repelling forces and stopping impulses (see [23]). If the distance between two approaching objects falls below a certain threshold, we apply a repulsion force. If the objects cannot be stopped in this way during the next few time steps, we apply stopping impulses, which reliably prevent imminent intersections. While this is a straightforward concept in the sequential case, there are some important implications for parallel implementations. We will discuss these issues in Sec. 5. Collision response to complex self-collisions often leads to secondary collisions, which also need to be handled to enable high quality simulations. If, after the first iteration of the collision handling phase, there are still remaining or newly introduced collisions, we handle those and do another collision detection step, until all collisions are resolved.

The output of the collision handling stage are new nodal velocities, which are then finally used to advance the system to the next time step, i.e., to compute new, collision-free nodal positions. This update is efficiently carried out in parallel using simple loop-level parallelism.

4. Parallel Solution of Sparse Linear Systems

In the following we assume a sparse linear system of the form $\mathbf{Ax} = \mathbf{b}$, which is to be solved numerically using the cg-method. The iteration stops when the norm of the residual has been decreased by a given factor (e.g., between 10^{-6} and 10^{-8} in our case) compared to the initial norm. The number of necessary iterations and therefore the speed of convergence depends on the condition number of the matrix $\mathbf{A}$. Usually, this condition number is improved using a preconditioning matrix $\mathbf{P}$ leading to a modified system

$$\mathbf{P}^{-1}\mathbf{Ax} = \mathbf{P}^{-1}\mathbf{b},$$

where $\mathbf{P}^{-1}\mathbf{A}$ is supposed to have a better condition number and $\mathbf{P}^{-1}$ is fairly easy to compute. The choice of an appropriate preconditioner is crucial because it can reduce the iteration count substantially.

The setup and solution of the linear system now breaks down to a sequence of operations in which (due to their computational complexity) the sparse matrix vector (spmv) multiplication and the application of the preconditioner are most important. Before we discuss these operations in more detail, we will first describe the underlying problem decomposition, which forms the basis for the actual parallelisation.

4.1. *Problem Decomposition*

As a basis for the following discussion, we assume the compressed row storage format for sparse matrices in which nonzero entries are stored in an array along with a row pointer and a column index array (see [6]). The most intuitive way to decompose the spmv-operation into a number of smaller sub-problems is to simply partition the matrix into sets of contiguous rows. The multiplication can then be carried out in parallel among the sets. This simple approach applies to the general sparse matrices. However, the matrices we deal with are always symmetric, which is due to the underlying partial differential equation. Hence, only the upper triangular part, including the diagonal, has to be stored. This leads to smaller memory requirements for the data as well as the index structure. The numerical kernel for symmetric matrices, which is described in Algorithm 1, is more efficient than the non-symmetric version (cf. [24]):

Algorithm 1 Symmetric Spmv-Multiplication

1: **for** $i = 1$ to n_{rows} **do**
2: $\quad start = ptr[i], end = ptr[i+1]$;
3: $\quad$ **for** $j = start$ to end **do**
4: $\quad\quad \mathbf{y}[i] += \mathbf{A}[j] * \mathbf{x}[ind[j]]$;
5: $\quad\quad$ **if** $i \neq ind[j]$ **then**
6: $\quad\quad\quad \mathbf{y}[ind[j]] += \mathbf{A}[j] * \mathbf{x}[i]$;
7: $\quad\quad$ **end if**
8: $\quad$ **end for**
9: **end for**

Here, *ptr* denotes the row pointer, *ind* refers to the pointer to the index structure, and $\mathbf{A}$ denotes the matrix data. Furthermore, $\mathbf{x}$ and $\mathbf{y}$ refer to the source and destination vector, respectively. The algorithm performs dot products between matrix columns and the source vector (line 4). During the same sweep through the matrix data, it also computes vector scalar products (or so called `axpy`-operations) between rows of the matrix and entries of the source vector (line 6). Hence, the algorithm visits every matrix entry only once, which gives rise to a computationally efficient implementation. However, a parallel implementation of the symmetric spmv-algorithm is more complicated: the access pattern to the solution vector is not as local as for the non-symmetric case since the entry which is written in line 6 can virtually be at any location in $\mathbf{y}$. The required synchronisation would make a direct parallel implementation of the symmetric spmv-kernel inefficient. Clearly, the simple row-based partitioning of the matrix $\mathbf{A}$ is not an adequate basis here.

Methods based on domain decomposition are better suited in this case. They divide the input data geometrically into disjoint regions. Here, we will only consider non-overlapping vertex decompositions, which result in a partitioning P of the domain Ω into subdomains Ω_i such that $\Omega = \cup_i \Omega_i$ and $\Omega_i \cap \Omega_j = \emptyset$, for $i \neq j$. Decompositions can be obtained using graph partitioning methods [25]. An example of this can be seen in Fig. 2, which also shows a special vertex classification. This will be explained in the next section.

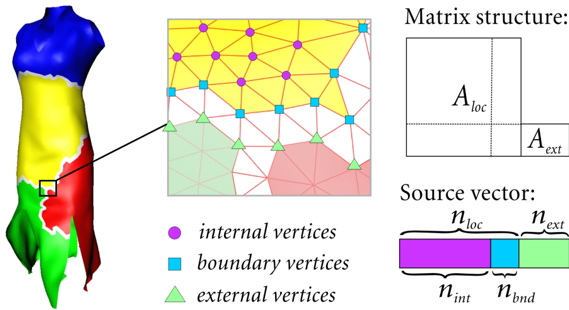

Fig. 2. Decomposition of a mesh into four disjoint partitions indicated by different colours. The vertex ordering and the resulting matrix structure for one of the partitions are shown to the right. The matrix $\mathbf{A}_{loc}$ describes the internal coupling of the nodes belonging to the partition (depicted as a square block). The dashed lines indicate the matrix entries corresponding to boundary vertices of the partition. The rectangular matrix $\mathbf{A}_{ext}$ describes the coupling between internal and external interface nodes.

4.2. *Parallel Sparse Matrix Vector Multiplication*

Let $n_{i,loc}$ be the number of local vertices belonging to partition i and let V_i be the set of corresponding indices. These vertices can be decomposed into n_{int} internal vertices and n_{bnd} interface or boundary vertices, which are adjacent to n_{ext} vertices from other partitions (see Fig. 2). If we reorder the vertices globally such that vertices in one partition are enumerated sequentially, we obtain again a partitioning of the matrix into a set of contiguous rows. The rows $a_{i,0}$ to $a_{i,n}$ of matrix $\mathbf{A}$ where $i \in V_i$ have the following special structure: the set of entries defined by $\{a_{lm} | l \in V_i, m \in V_i\}$ forms a symmetric submatrix $\mathbf{A}_{i,loc}$ lying on the diagonal of $\mathbf{A}$. The nonzero entries in this block describe the interaction between the local nodes of partition i. More specifically, this means that when nodes l and m are connected by an edge in the mesh, there is a nonzero entry a_{lm} in the corresponding submatrix of $\mathbf{A}$. Apart from this symmetric block on the diagonal there are further nonzero entries a_{le} where $l \in V_i$ is an interface node and $e \notin V_i$. These entries describe the coupling between the local interface nodes and neighbouring external nodes.

The matrix vector multiplication can be carried out efficiently in parallel if we adopt the following local vertex numbering scheme (cf. [6]). The local vertices are reordered such that all internal nodes precede the interface nodes. For further performance enhancement, a numbering scheme that exploits locality (such as a self avoiding walk [9]) can be used to sort the local vertices. Then, external interface nodes from neighbouring partitions are locally renumbered as well. Let $\mathbf{A}_{ext}$ be the matrix which describes the coupling between internal and external interface nodes for a given partition. Notice that $\mathbf{A}_{ext}$ is a sparse rectangular matrix with n_{bnd} rows. With this setup the multiplication proceeds as follows:

(i) $\mathbf{y}(0, n_{loc}) = \mathbf{A}_{loc} \cdot \mathbf{x}(0, n_{loc})$

(ii) $\mathbf{y}(n_{int}, n_{loc}) = \mathbf{y}(n_{int}, n_{loc}) + \mathbf{A}_{ext} \cdot \mathbf{x}_{ext}(0, n_{ext})$

The first operation is a symmetric spmv-multiplication, the second one is a non-symmetric spmv-multiplication followed by an addition. Both these operations can be carried out in parallel among all partitions. This decomposition is not only used for the spmv-kernel but also as a basis for the parallel matrix assembly as well as for the parallel preconditioner, which will be presented next.

4.3. *Parallel Preconditioning*

In order to make the cg-method fast, it is indispensable to use an efficient preconditioner. There are a broad variety of different preconditioners ranging from simple diagonal scaling (Jacobi preconditioning) to sophisticated multilevel variants. For an actual choice one has to weigh the time saved from the reduced iteration count against the cost for setup and repeated application of the preconditioner. Additionally, one has to take into account how well a specific preconditioner can be parallelised. Unfortunately, designing efficient preconditioners is usually the most difficult part in the parallel cg-method [7]. As an example, the Jacobi preconditioner is very simple to set up and apply, even in parallel, but the reduction of necessary iterations is rather limited. Preconditioners based on (usually incomplete) factorisation of the matrix itself or an approximation of it are more promising. One example from this class is the Symmetric Successive Overrelaxation (SSOR) preconditioner. It is fairly cheap to set up and leads to the solution of two triangular systems. For the sequential case, this preconditioner has proven to be a good choice in terms of efficiency [26]. However, parallelising the solution of the triangular systems is very difficult. Even if it is not possible to decouple the solution of the original triangular systems into independent problems we can devise an approximation with the desired properties. Let $\bar{\mathbf{A}}$ be the block diagonal matrix with block entries $\mathbf{A}_{ii} = \mathbf{A}_{i,loc}$ (see Fig. 2). Visually, the external matrices $\mathbf{A}_{ext}$ are dropped from $\mathbf{A}$ to give $\bar{\mathbf{A}}$. Setting up the SSOR-preconditioner on this modified matrix leads again to the solution of two triangular systems. However, solving these systems breaks down to the solution of decoupled triangular systems corresponding to the $\mathbf{A}_{i,loc}$ blocks on the diagonal. This means that they can be carried out in parallel for every partition.

Approximating $\mathbf{A}$ with $\bar{\mathbf{A}}$ means a loss of information, which in turn leads to an increased iteration count. The actual overhead resulting from this approximation depends on the number of partitions used for the problem decomposition. Hence, the increased parallelism has to be weighed against this additional overhead when deciding on an actual number of partitions. For our experiments we generally set the number of partitions equal to the number of cores available (see subsequent discussion). For this specific choice the incurred overhead remains small compared to the speedup obtained through parallelisation.

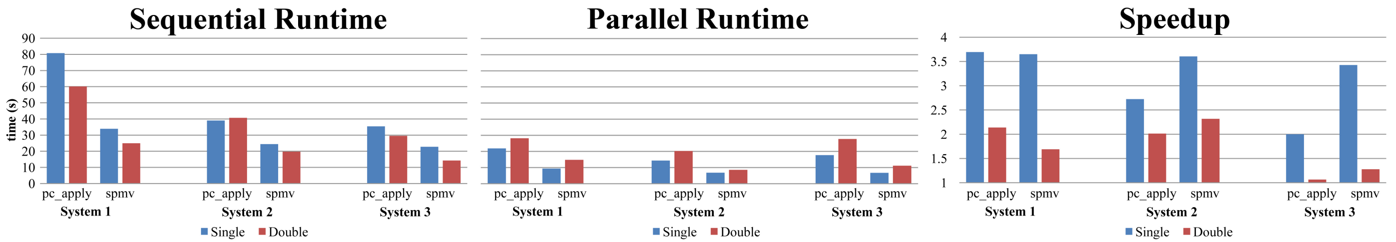

Fig. 3. A comparison of performance obtained for single and double precision arithmetic. The diagram shows sequential runtimes (*left*), parallel runtimes (*middle*), and corresponding speedups (*right*) obtained for preconditioner application (pc_apply) and sparse matrix vector multiplication (spmv) on the three test platforms.

4.4. *Parallel Efficiency Considerations*

There are further important aspects that have to be taken into consideration in order to set up an efficient parallel implementation of the cg-method. Dense matrix multiplications usually scale very well since they have regular access patterns to memory and a high computational intensity per data reference. For the spmv-kernel, however, the picture is quite different. Considering the structure of this kernel (see Algorithm 1), it can be seen that the actual matrix data, as well as the index structure, are traversed linearly while accesses to the data of the source vector and the data of the destination vector occur in a non-contiguous fashion, i.e., the locality of these data accesses cannot be assumed. The performance of the spmv-algorithm is therefore mostly limited by memory latency and bandwidth, as well as cache performance. As a result, only a fraction of the theoretical peak performance for floating point operations can be attained for the cg-method on a given machine. According to Vuduk et al. [27] this fraction is often less than 10%. This dependence is even more pronounced for multi-core processors, where typically two or more cores share a single memory interface. It is therefore important to improve data locality and thus cache performance. One way to achieve better locality is to exploit the natural block layout of the matrix as determined by the underlying partial differential equation: the coupling between two vertices is described by a $3{\times}3$ block – therefore nonzero entries in the matrix always occur in blocks. Using a block data layout already leads to a significant improvement. Additional benefits can be achieved using single precision floating point data instead of double precision. This reduces the necessary matrix data (not including index structure) transferred from memory by a factor of two. In order to determine the resulting performance benefit experimentally, we use a simple test case as a synthetic benchmark. The input for this test is a flat elastic surface with very high resolution (91,200 vertices). The surface is first isotropically stretched by a factor of 1.05 and then released at the beginning of the simulation. For simplicity, gravity was turned off for this test. Since the surface remains perfectly flat, collisions do not occur and collision handling was therefore deactivated. We used four partitions for the problem decomposition and four threads for parallel computations. In order to reduce the overall computation time we simulated only 1/30 seconds in this test. Nevertheless, this period is long enough to capture the numerical performance since it includes roughly 1300 preconditioner applications and spmv-operations.

The results of this experiment are summarised in Diagram 3 (see Appendix A for a detailed description of the systems used). The measurements for the sequential runs already reveal some interesting aspects. The application of the preconditioner is computationally more expensive than the spmv-operation on all three systems. This is not surprising since both operations are called in every iteration of the cg-method and one preconditioner application, involving the solution of two triangular system, is more expensive than one spmv-operation. The diagram also shows that the performance difference between single and double precision is not very pronounced. This rather unexpected behaviour could be the manifestation of very effective latency hiding by the CPUs and the compiler. Moreover, neither alternative can consistently outperform the other on all three systems. Considering the parallel timings, however, we can conclude that the single precision variant performs consistently better on all systems. The corresponding speedup values are shown on the right in Diagram 3. Depending on the actual system, a performance gain between 2.0 and 3.7 for the preconditioner application and even 3.4 to 3.7 for the spmv-operation can be obtained for single precision. The speedup values for double precision are not as good on all three systems and, in particular, there is virtually no speedup for system 3. We conjecture that this is due to the fact that the four cores of system 3 share a single memory interface.

In conclusion, the parallel timings obtained for this experiment suggest that single precision arithmetic should generally be preferred over double precision. In consequence, the remaining investigations as well as the numerical experiments are solely based on single precision arithmetic.

In the following section we will discuss some implementation related issues, which have to be considered in order to implement the single precision variants of the algorithms in a stable way.

Using Single Precision Floating Point Data In engineering applications absolute accuracy is often of utmost importance. It is therefore mandatory to use double precision arithmetic for the numerical algorithms resulting from

finite element discretisations. For the case of physically-based simulation in computer graphics, however, visual quality is usually more important than absolute numerical accuracy. For our cloth simulator, we found that, with only minor modifications, even the largest examples led to no problems using single precision arithmetic. There are, however, a few pitfalls to avoid. The critical arithmetic operations where accuracy might potentially be lost are not divisions or multiplications but additions and subtractions where so called cancellation occurs. This can be illustrated with a simple example: Consider the following sequence of summations

$$a = b + c, \quad d = a - c,$$

where $b = 10^{-7}$ and $c = 10$. Using a 32-Bit single precision data type for the variables involved, the numerical result is $d = 0$ where it should be $d = 10^{-7}$. This behaviour can be explained by considering the definition of this data type. Assuming the IEEE754 standard, a single precision floating point value consists of 1 bit sign s, 23 bit mantissa m and 8 bit exponent p, leading to the representation $f = (-1)^s \cdot m \cdot 2^p$ (see the standard for more precise definitions). Therefore, if, for a large summand $f_1 = m_1 \cdot 2^{p_1}$ and a small summand $f_2 = m_2 \cdot 2^{p_2}$, it holds that

$$2^{-23} \cdot 2^{p_1} > m_2 \cdot 2^{p_2} \Leftrightarrow p_1 - p_2 > 23,$$

then the small summand will be absorbed in the summation, i.e. $f_1 + f_2 = f_1$. Generally, such situations cannot be avoided in advance since the relative sizes of the summands are not known. However, in many cases it is either possible to reorder summations by hand or to resort to double precision floating point data for intermediate results. In our implementation, for example, we reordered arithmetic operations for the singular value decomposition of a 2×2 matrix necessary for extracting rotations from the displacement field (see [19]). Furthermore, we eliminated accuracy concerns in the computation of the bending forces, involving trigonometric functions and square root operations (see [28]), by using double precision for intermediate results. Another example is the computation of internal forces which can be done as $\mathbf{f} = \mathbf{Ku}$, where $\mathbf{K}$ is the stiffness matrix and $\mathbf{u} = \mathbf{x} - \mathbf{x}_0$ is the vector of nodal displacements defined as the difference between current positions $\mathbf{x}$ and rest positions $\mathbf{x}_0$. For technical reasons we originally separated this expression into the sequence $\mathbf{f} = \mathbf{Kx} - \mathbf{Kx}_0$. Although convenient when using double precision, we found that this degrades accuracy when using single precision floating point arithmetic. We therefore rearranged this expression and compute the displacement $\mathbf{u}$ first, before carrying out matrix multiplication.

In summary, switching to single precision arithmetic requires only a few modifications to yield a stable implementation. In particular, we did not encounter any accuracy degradations or convergence problems for the solution of the linear system of equations.

Influence of the Number of Threads and Partitions When setting up the problem decomposition for solutions of the linear system of equations, the intuitive choice for the number of partitions is to use as many as there are cores available in the parallel system. For distributed memory systems this one-to-one correspondence is often the only practically realisable choice. Furthermore, a partition count exceeding the number of nodes in a cluster would result in an increase in communication, which is usually prohibitively expensive in the context of distributed memory computing.

Shared memory parallel systems are not as restrictive in terms of communication costs such that using a higher number of partitions can be considered. In fact, memory latency may actually be hidden using more partitions than available cores. Additionally, it may be beneficial to adjust the number of partitions in such a way that the data (or working set) for a single partition fits into the cache of processors. We tested the influence of this parameter experimentally using the synthetic benchmark described above. The performance differences were, however, rather small and no improvements could be obtained by using partition counts higher than four.

Besides the number of partitions, a further parameter that can be considered in this context is the number of threads used for parallel computations. Again, it seems an obvious choice to use as many threads as there are partitions. However, provided that the number of threads remains within reasonable bounds (say two to four times the number of cores available in the system), we should at least not expect a negative impact. In order to determine the influence of the number of threads experimentally, we used again the synthetic benchmark. This time we varied both the number of partitions as well as the number of threads. For the sake of conciseness, we only show the results obtained for the four partition case (see Diagram 4).

Again, the differences are only marginal and no advantages result from using a higher number of threads than cores available. In summary, we can state that it does not pay off to increase the number of partitions or threads.

5. Parallel Collision Handling

From the parallelisation point of view, the collision handling stage differs substantially from the problem of implicit numerical time integration. Collisions can be distributed very unevenly in the scene and their typically changing locations cannot be determined statically. This is why the naive approach of letting each processor handle the collisions of its own partition can lead to considerable processor idling, which seriously affects the overall parallel efficiency. Therefore, a dynamic problem decomposition is mandatory. Our basic parallelisation strategy is similar compared to previous work aimed at distributed memory architectures [16]. However, the shared-memory setting enables us to set up heuristics exploiting temporal and spatial coherence. In this way, we can effectively control thread creation overhead.

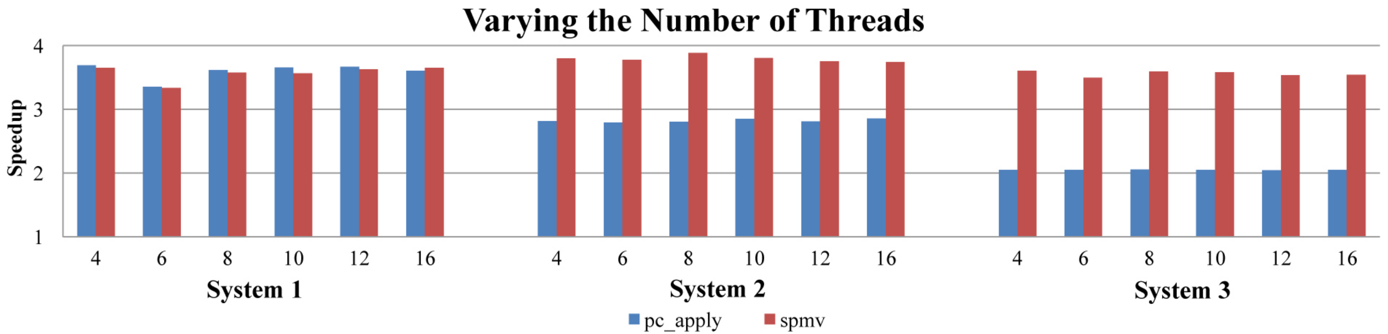

Fig. 4. Influence of the number of threads used for parallel computations. Values on the vertical axis refer to speedup while values on the horizontal axis denote the number of threads.

5.1. *Basic Problem Decomposition*

The recursive collision test of two bounding volume hierarchies can be considered a case of depth-first tree traversal. For inducing parallelism, we implemented this procedure using a stack which holds individual tests of two bounding volumes. During the traversal, the expansion of a node yields n additional child nodes. We process one node immediately while the others are pushed onto the stack. The traversal proceeds downwards until a leaf is reached. Upward traversal begins by processing elements from the stack. In this way, all of the nodes in the tree are visited. The basic idea for dynamically generating parallelism is to now remove nodes from the stack in an asynchronous way and to create tasks from them. One or more tasks can then be assigned to a thread and executed on an idle core.

Unlike in the distributed memory setting, we do not have to consider load balancing explicitly. As long as there are enough threads ready for execution the scheduler of the operating systems will keep all cores busy. However, for problems with high irregularity, like parallel collision handling, it is generally impossible to precisely adjust the amount of logical parallelism to be exploited to the amount of available parallelism, i.e., idle processors. Thread creation overhead can contribute considerably to the overall parallel overhead, especially on shared memory architectures. Therefore, an over-saturation of threads has to be avoided as well.

In our approach, we minimise thread creation overhead on two levels. At the algorithmic level, we employ an heuristics-based approach which prevents threads with too fine a granularity from being generated. At the implementation level, we decouple the process of thread creation and thread execution by using an execution model based on a task-pool. Specifically, we employ lock-free data structures for implementing the task pool such that thread execution overhead is further minimised. The next two paragraphs explain these optimisations in more detail.

5.2. *Controlling Task Granularity*

To effectively control the granularity of a task, we need a good estimate of how much work corresponds to a certain task. The computational cost for carrying out a test in the collision tree is determined by the number of nodes in its subtree. Generally, this number is not known in advance. Because of the temporal locality inherent in dynamic simulation we can, however, exploit coherence between two successive time steps. During each collision detection pass we keep track of the number of tests in the respective subtree for every node in the collision tree using back propagation. This information is then used as a work estimate for tasks in the subsequent collision handling phase. Thus, the computation of the work estimate is fully integrated into the collision detection phase and its overhead is well below the measurement threshold.

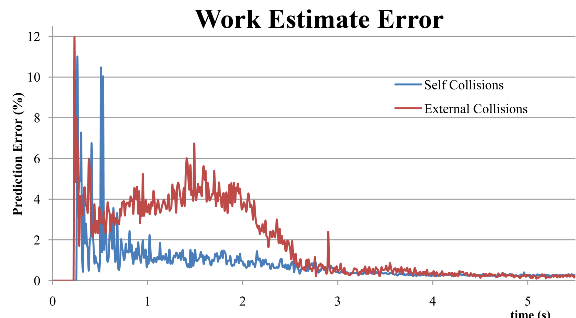

Fig. 5. Work estimate error for scene 2. The diagram shows the deviation from the actual amount of work over time as a percentage. Even in this very dynamic scene, the temporal coherence is high.

In this way, we can avoid creating tasks with too small an amount of work. Additionally, we can use this information to determine which tests should be carried out immediately. The error involved in the work estimation is usually very small. This can be seen in Fig. 5, which shows the error made by our work predictor for 5.5 seconds of simulation of our second test scene (refer to Appendix B for a detailed description). Even though this is a highly dynamic scene, the prediction error quickly drops to less than 5% and after a short while stabilises below 1%. For the test scenes (1a) and (1b), the error drops even faster, since these scenes are far less dynamic. After an initial phase where the cloth settles on the avatar, the error falls below 1% for the rest of the simulation, for both external and self-collisions.

To evaluate the benefit arising from this new task splitting scheme, we performed comparisons with two alterna-

tive approaches. The first one, being the simplest variant, carries out the test corresponding to the leftmost subnode immediately and assigns the remaining subnodes to tasks. The second one is based on randomisation, which is a widely adopted paradigm for achieving well-balanced load distribution in parallel applications. In this case, we randomly select the subnode to be treated immediately.

The results of these comparisons are visualised in Fig. 6, which shows that our new scheme is very competitive. While the randomised variant performs similarly to the simple approach, our work estimation scheme can improve on this. We found that it yields an improvement even in scenes with a high number of collisions like test scene (1a). In this scene, the high number of collisions is due to the dense meshes, and it is thus relatively easy to schedule the tasks efficiently. Even the simple and the random scheme achieve speedups of about 3.5. Using our work estimate, we can slightly improve on this and bring the speedup to a solid 3.6 on all three test systems. The overhead for the computation of the work estimate is negligible and already included in the speedup shown in the diagrams. A greater improvement from the optimised task creation strategy can be seen in test scene (1b), which has a smaller number of nodes. Since, in this case, the number of collision events is considerably lower than in the almost saturated high polygon count scene, it is more important to avoid creating tasks with little or no work (see Fig. 6). Neither the simple nor the random scheme are able to attain a speedup of greater than 3.0, while the coherence-based scheme easily achieves 3.1 to 3.4 on the different test systems.

More challenging is test scene (2) with its highly dynamic movements. Even though there is less temporal and spatial coherence to exploit than in test scenes (1a) and (1b), our work estimate can still improve on the random scheme for test systems 1 and 2. The simple and random schemes only yield a speedup of slightly greater than 3.0, while the coherence scheme achieves about 3.3. The third test system shows a very good speedup of 3.5 with the random scheme, and a still very good one of 3.4 with the coherence scheme. All in all, our new work predictor seems to be able to improve on the simple and random scheme in almost all test cases, and even in those where it does not improve the speedup, it still delivers a performance that is little worse than the simpler approaches. The work estimate predictor was employed for all measurements of the test scenes discussed in Sec. 6.

Ground Truth To verify that our work estimate is close to optimal, we compared the speedup obtained with our predictor to the one obtainable with a ground truth data set. We computed this ground truth data using an additional pass. During this first pass, the precise number of collision events for the subtree under each inner node in the collision test tree was written out to disk. In a second pass, we used this information instead of our prediction. Thus, our algorithm could make the decision of whether or not to create

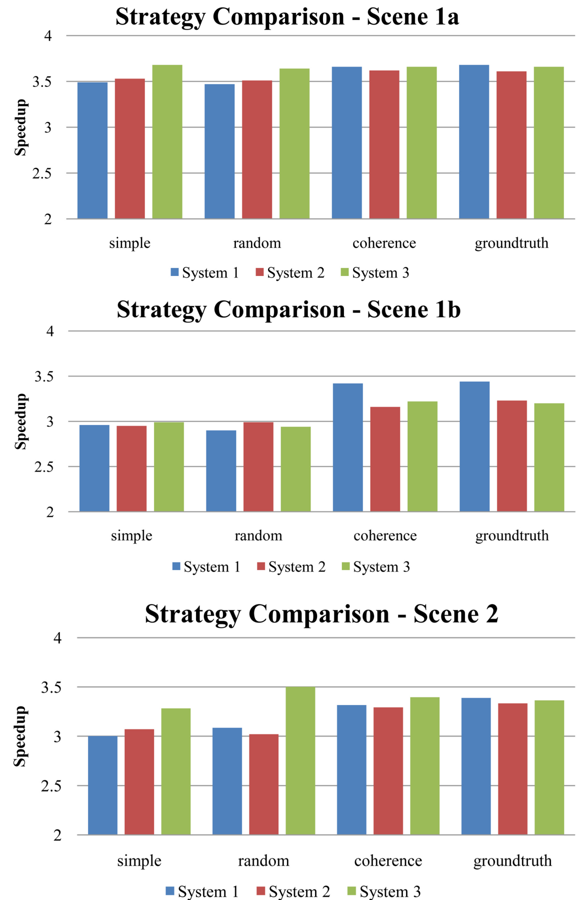

Fig. 6. Comparison of different strategies used for task generation in scenes (1a), (1b) and (2). The overhead for the computation of the work estimate is included.

a task based on the actual number of collisions that would occur in this time step, instead of a prediction based on the number of collisions in the last time step. The results are shown in Fig. 6. They clearly show that the prediction is almost as good a decision factor as the true number of collision events obtained from the ground truth dataset. This attests to the fact that temporal coherence in dynamic collision detection is a valuable source for performance improvement.

5.3. *Implementation*

As in our previous work, which addressed distributed memory architectures, we employed the DOTS system platform [29] for parallelising collision handling.

DOTS especially supports highly irregular task parallel applications by means of the multithreading programming model (not to be confused with the shared-memory model). The DOTS programming model is an extension of the Cilk model [30] designed to support shared and distributed memory architectures.

In [3] the interested reader will find a detailed description of how we have modelled the fully dynamic problem decomposition process (see Sec. 5.1) with the strict multi-

threading parallel programming model provided by DOTS.

In this section we discuss how we have modified the core of the run-time system of DOTS in order to further optimise the execution process on shared-memory architectures. Our main design goal was to minimise thread execution overhead while at the same time keeping all specific functionality of DOTS required to efficiently support highly irregular applications, e.g., the decoupling of thread creation and execution or control of non-determinism.

Basically, DOTS employs lightweight mechanisms for manipulating threads. Forking a thread results in creating a (passive) thread object, which can later be instantiated for execution. Thread objects are either executed by a preforked (OS native) worker thread or can be executed as continuation of a thread that would otherwise be blocked, e.g., a thread reaching a synchronisation primitive.

The run-time system of DOTS is based on a task pool execution model. When a thread is spawned (by the dots_fork primitive), the corresponding thread object is placed into a task pool, which is basically a queue data structure. On program startup, a worker thread is created for each core in the system. Worker threads take thread objects out of the task pool and process their run method. In our case the run method executes the bounding volume hierarchy testing procedure, as discussed in Sec. 5.1. Upon completion of a thread, the corresponding thread object (including the result of the thread) is placed into a second data structure called the ready queue. Subsequently, the worker thread gets the next thread object from the task pool and executes it. The dots_join primitive removes thread objects from the ready queue and delivers the result of the corresponding thread to the calling thread. Note that the runtime system of DOTS is only active during the task-parallel collision handling phase. When program execution is outside collision handling, the task pool and the ready queue are empty and the worker threads are suspended.

In a shared-memory setting, the task pool and the ready queue are both concurrently accessed by several threads. The original version of DOTS used mutual exclusion locking primitives for ensuring the consistency of the shared data structures. In our new approach the task pool and the ready queue are implemented using lock-free techniques [31].

Lock-free synchronisation is based on an atomic update operation, which must be supported by the processor. This operation, commonly referred to as CAS (compare and swap), atomically updates a memory location provided its initial content has some expected value. If the value is different, the update fails. For example, on the x64 architecture family the `lock cmpxchg16b` instruction provides appropriate CAS functionality.

Basically, lock-free techniques employ the CAS operation to realise an optimistic approach for ensuring consistency of concurrent data structures. A thread loads a value from a shared memory location and stores it in a local variable. The thread now performs a calculation on the local copy resulting in a new value. Finally, the thread tries to update the original memory location with the CAS instruction, supplying the original and the updated value for comparison. If the content of the shared memory location has not been changed, the update succeeds and the algorithm can proceed. It fails, however, if another thread has updated the shared-memory location during the calculation of the new value. In this case, the memory location is re-read and the update is tried again with a newly calculated value, until the CAS operation succeeds.

In the case of lock-free queues, the shared-memory location is typically a pointer variable, which is part of a linked list. This pointer is updated in order to insert or remove a list element. In this context, a subtle problem can occur when applying the described techniques. When a memory location is reused after the corresponding element has been removed from the list the same pointer again becomes part of the list but now represents a different element. However, this change cannot be detected by the CAS operation since the two elements are represented by the same pointer. Thus, when such a situation occurs during another update, the previous update will be lost. This issue is commonly referred to as the ABA problem. To avoid the ABA problem, reference counters can be associated with pointer variables, which are both compared by the CAS operation. In [32] further details on the implementation of a lock-free queue data structure are discussed.

Compared to the mutual exclusion approach, lock-free programming reduces overheads since no system calls are needed for acquiring and releasing locks. Lock-free data structures also scale well to a larger number of processors/cores since they considerably reduce contention.

We conducted performance measurements to compare our new lock-free approach with the original implementation based on mutual exclusion locks. To assess the efficiency of the implementations we determined the execution overhead of DOTS threads, which is the execution time of a DOTS thread performing no computation.

Our first test series focuses on latency aspects of thread execution, which is a measure of the isolated thread execution overhead. In the corresponding test program one core forks a thread and immediately joins it. The thread is executed by another idle core. For computing the mean thread execution overhead this fork-join sequence is repeated 1,000,000 times. The second test series investigates throughput aspects of thread execution, which indicates the granularity of thread execution. Here, one core forks 1,000,000 threads at once, which are concurrently executed by the remaining cores. Fig. 7 shows the resulting mean thread execution overhead for the discussed tests executed on our test systems 2 and 3 (see Appendix A for a detailed description). On system 1 the `lock cmpxchg16b` instruction is not available. However, it has been added on subsequent processor generations of the AMD64 architecture, see [33]. Thus, the described lock-free techniques cannot be realised on system 1. For all measurements presented later in this paper the original version of DOTS (based on mutual exclusion locks) has been used for system 1 .

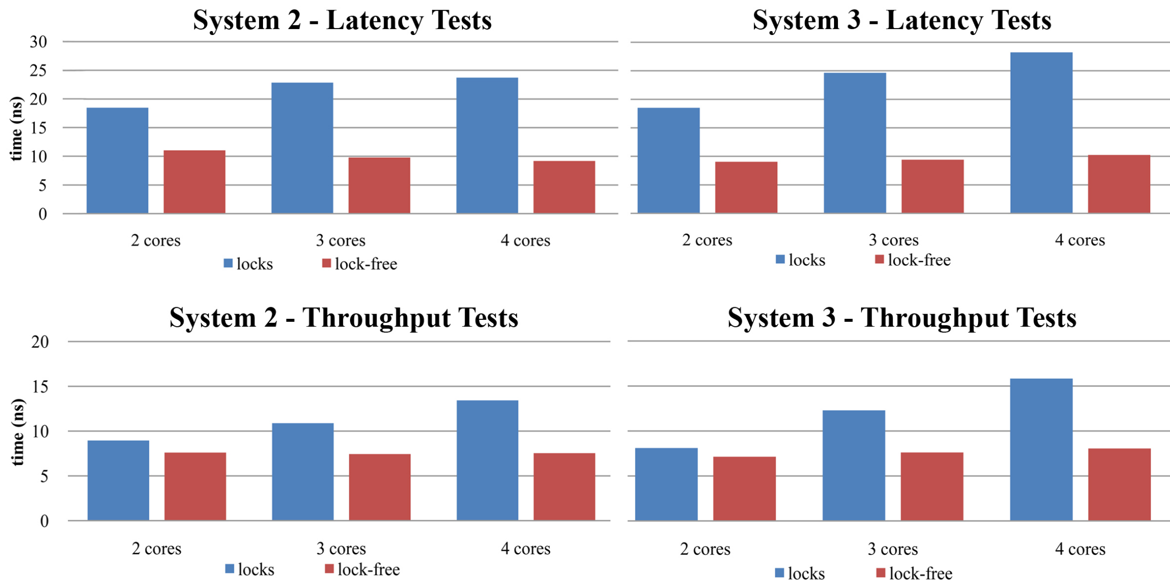

Fig. 7. Thread execution overhead for latency and throughput oriented test scenarios.

For all settings the execution overhead of DOTS threads is significantly lower when our new lock-free implementation is used. Moreover, with lock-free techniques the overhead increases only slightly for a larger number of cores, revealing a considerably improved scalability.

In our latency oriented test scenario, threads are more frequently blocked and unblocked than in the throughput oriented tests. Thus, OS overhead for thread management represents a substantial fraction of the execution overhead of DOTS threads for the version based on mutual exclusion locks. Moreover, the throughput oriented test scenario enables parallel execution of DOTS threads, which further reduces the measured execution overhead. This holds also for the lock-free version. These two observations explain the higher execution overhead of the latency tests compared to the throughput oriented tests.

When the results for 4 cores are compared on the two systems one can see that system 2 performs better than system 3. This can be explained by the limited scalability of the memory interface of system 3. This effect is especially pronounced for the implementation based on locks, since it involves considerably more instructions and thus more memory accesses.

6. Results

This section presents the results of the experimental studies used to evaluate the methods presented in this work. The tests were run on three different platforms, which are desribed in Appendix A. Detailed descriptions of the test scenes can be found in Appendix B. Separate timings are given for the three important phases, i.e., application of the preconditioner (*pc_apply*), sparse matrix vector product (*spmv*) and the collision handling stage (*collision handling*). All results are averaged from three test runs, and all simulations were run using single precision floating point arithmetic.

Scene 1. The left plot in Fig. 8 shows the results obtained for the first version of scene 1, indicating a high parallel efficiency for all stages. The speedup stays nearly constant over time for this rather static scene, as can be seen in the leftmost plot of Fig. 9. Due to the high number of collisions that occur, a convincing sustained speedup of about 3.6 is attained for the collision handling phase. The speedup over time is plotted only for measurements on test system 1, since the plots of the other test systems are very similar and exhibit the same features. The only noticeable differences are that the respective plot curves are shifted slightly up or down, as compared to the speedup on test system 1. These offsets can be found in Fig. 8, by comparing the speedups of the other test systems to the first one.

The centre diagram of Fig. 8 and the centre plot of Fig. 9 show the speedup for the second version of scene 1, using a much lower resolution for the avatar. This reduces the number of collisions drastically, making it more difficult to achieve good speedups. Still, our work estimate keeps the parallel workload well distributed and achieves a speedup of well over 3.2 on the Intel-based systems 2 and 3, and 3.4 on the Opteron-based system 1. Refer to Fig. 6 for a comparison of the different work distribution strategies for all the test scenes. Although the resolution of the textile is unchanged, the numerics behave differently. This is most likely due to changed cache utilisation, since the total problem size is much smaller than in the first version. This hypothesis is supported by the fact that test system 1, with the smallest L2 cache, yields much better results with the smaller version of the test scene, up to a close-to-ideal speedup of 3.9 for the application of the preconditioner.

The evolution of the speedup over time is not as constant as with the high resolution version (1a). There is a peak

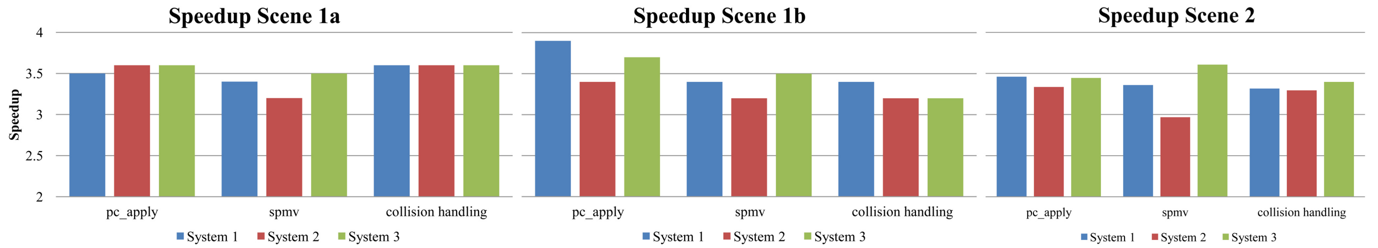

Fig. 8. Integral speedups obtained for test scenes (1a), (1b) and (2). The diagrams show the average speedups of the different stages for both numerics and collision handling.

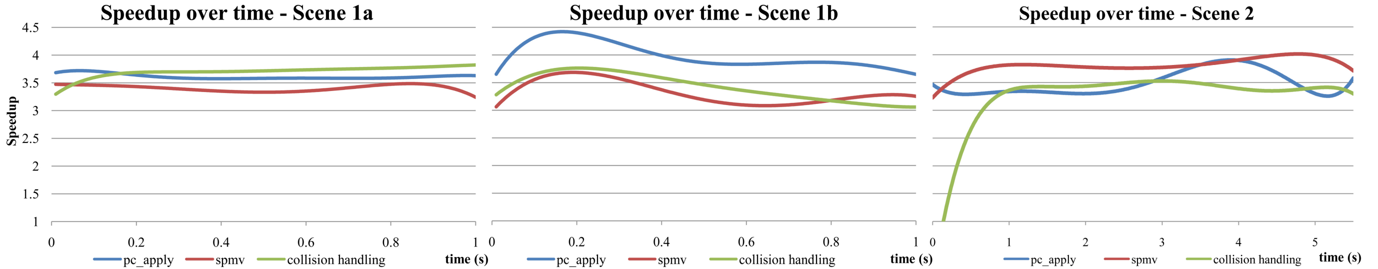

Fig. 9. The plots shows the evolution of the speedups for numerics and collision handling over time, measured for scenes (1a), (1b) and (2) on System 1.

during the first 300ms of simulation, which corresponds to the cloth settling on the avatar. This initial phase of the simulation has a high amount of work for both stages of the simulation, both numerics and collision handling, and is thus easier to distribute efficiently. Afterwards, the cloth has settled down on the avatar, and both the iteration count of the cg-method, as well as the number of handled collisions, goes down, making it harder to find enough work to keep the cores busy.

Scene 2. The rightmost plot in Fig. 8 shows the results for scene (2). Even though this scene exhibits much more complicated collisions than the first test scene, particularly with respect to self-collisions, the speedup of about 3.3 obtained for the collision handling stage is not as good as for the high resolution version of the first test scene. This is due to the fact that scene (2) is much more dynamic, and the collisions are distributed more irregularly, making it harder to schedule them efficiently. Our work predictor still manages to keep the speedup at a good level of 3.4, but for this scene we do not improve on the randomised approach for all test systems. Still, as can be seen from Fig. 6, the difference is not large and we feel that it is practically more relevant to achieve good speedups on scenes like (1a) and (1b). The results are also much more consistent when using our work predictor: all three systems exhibit speedups of about 3.3, while with the randomised approach the range is from 3.0 for test system 2 to 3.5 for test system 3.

The evolution of the speedup over time plotted in Fig. 9 shows a steep increase in the collision handling speedup during the first second. At the beginning of the simulation, no collisions occur at all, and then as the ribbon falls onto the inclined plates their number increases rapidly, until after about 1s of simulation time the speedup is at its peak of 3.5. The numerics speedups stay mostly constant, with a small peak after about 4s, which corresponds to the ribbon hitting the ground plate. During this impact, the ribbon experiences higher deformations leads, which, in turn, results in a slightly increased iteration count of the cg-method

7. Conclusions and Future Work

In this work we have presented key techniques for parallel physically-based simulations on multi-core architectures. We focused on the two major bottlenecks of the simulation, namely the solution of the linear system and the collision handling stage, and proposed efficient parallel algorithms to accelerate these problems. Our performance measurements confirm the parallel efficiency of these methods and indicate that physically-based simulations on modern commodity platforms can be greatly accelerated if parallelism is exploited. Because the scalability is encouraging, we would like to further explore these methods using more processors. Furthermore, we envisage transferring our framework to a hierarchically structured parallel environment, in which the nodes of a distributed memory cluster are each symmetric multi-processor machines with multiple cores.

8. Acknowledgements

The second author was supported by DFG grant STR 465/21-1. The third author was supported in part by the Ohio Supercomputer Center. We also thank our reviewers for their constructive critique.

Appendix A. Test Systems

Because the aim of this work is to accelerate computations for physically-based simulations on commodity platforms, we decided to use systems that are easily available at the current time. In order to show that our approach is general and not limited to a specific type of multi-core system, we carried out the tests on three different systems. All computers are equipped with 4 CPUs of similar clock speeds, but differ in their approach to multi-processing. Comparing the scaling on three distinct platforms should create useful data points, as compared to just analysing a single platform.

#	Architecture	Cores	L2 Cache	RAM
1	AMD Opteron	2×2@2.00Ghz	4×1MB	2GB
2	Intel Xeon	2×2@2.33Ghz	2×4MB	4GB
3	Intel Core2	1×4@2.40Ghz	2×4MB	2GB

Table A.1
Systems used for performance experiments

System 1. The first system is based on a dual AMD Opteron 270 (Italy) machine with 2GB of main memory. Each of the Opterons is a dual core processor running at 2.0GHz. The memory architecture is shared address-space, more specifically cc-NUMA (cache-coherent-NUMA). Each Opteron core has 1MB of dedicated L2 cache.

System 2. The second platform is a dual Intel Xeon 5140 (Woodcrest) running at 2.33GHz and equipped with 4GB of main memory. Again, each of the CPUs is a dual core processor, but each core has access to a much larger unified L2 cache of 4MB. It is based on a classic SMP (symmetric multi-processor) system, but has an additional memory bus. This means that both of the dual-core processors have their own dedicated memory interface.

System 3. The third test platform is an Intel Core 2 Quad Q6600 (Kentsfield) running at 2.40GHz equipped with 2GB of main memory. Again, each core has access to 4MB of unified L2 cache, for a total of 8MB on die L2 cache. Compared to the second system an important difference is that the four cores share a single memory interface. As a result, the total memory bandwidth is approximately half that of the Woodcrest-based system. This characteristic may have a negative impact on the overall performance when it comes to memory-intensive applications.

Appendix B. Test Scenes

We tested our approach with two scenes, which highlight different aspects of the simulation. Since the focus is on accelerating commonly used scenarios, we decided to use only moderately large input data. This is an important difference to the distributed memory setting, which traditionally aims at problem sizes exceeding the capacity of a single workstation.

Scene 1. The first example (1a) (see Fig. B.1) is a simulation of a dress worn by a female avatar with a fairly complex geometry (over 27,000 vertices). The dress, consisting of roughly 4,500 vertices, is pre-positioned around the body and drapes under gravity during one second of simulation. This test scene focuses primarily on the parallel performance of the numerical time integration and on cases with mostly evenly distributed collisions. Most of the collisions in this scene occur between the avatar and the dress, with only very few self-collisions in the lower part of the dress. A second version (1b) of this test scene includes a lower resolution avatar (roughly 1,800 vertices) and is otherwise identical to (1a). It serves as a test of how well our approach copes with scenes of lower resolution, where less work is available for distribution to the available cores.

Relatively static scenes like this are typical for virtual-try-on scenarios and for applications in the apparel industry, e.g. as a visualisation step during clothing design in a CAD system. Temporal and spatial coherence are high and can easily be exploited to speed up the simulation.

Scene 2. The second test scene is much more dynamic and puts special emphasis on collision handling. The deformable object is a long vertically oriented ribbon, comprised of 4,141 vertices (see Fig. B.2). It first falls onto two differently inclined planes, from which it rebounds towards the floor, where it finally comes to rest. The planes and the floor consist of 13,600 vertices in total. In the course of the simulation, external collisions as well as complicated self-collisions occur. The collisions are, however, not as evenly distributed as in the first example and change dynamically over time. Hence, the temporal and spatial coherence is considerably lower than in the first scene. Another complication is that many multi-collisions occur, forcing the collision handling to iterate several times until a collision-free state is attained. Multi-collisions arise when many textile layers are in close proximity, so that handling one collision causes new secondary collisions. Thus, this very dynamic test scene, with its many and irregularly distributed collisions, serves as an extreme challenge for the parallel collision handling stage of our approach.

References

[1] D. Baraff, A. Witkin, Large steps in cloth simulation, in: Proceedings of ACM SIGGRAPH '98, 1998, pp. 43–54.

[2] J. R. Shewchuk, An introduction to the conjugate gradient method without the agonizing pain, Tech. Rep. CS-94-125, Carnegie Mellon University, Pittsburgh, PA, USA (1994).

[3] B. Thomaszewski, W. Blochinger, Physically based simulation of cloth on distributed memory architectures, Parallel Computing 33 (6) (2007) 377–390.

[4] B. Thomaszewski, S. Pabst, W. Blochinger, Exploiting parallelism in physically-based simulations on multi-core processor architectures, in: Proc. of Eurographics Symposium on Parallel Graphics and Visualization, 2007, pp. 69–76.

[5] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory (2004).

[6] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd Edition, SIAM, 2003.

[7] J. Demmel, M. Heath, H. van der Vorst, Parallel numerical linear algebra, in: Acta Numerica 1993, Cambridge University Press, Cambridge, UK, 1993, pp. 111–198.

[8] D. O'Hallaron, Spark98: Sparse matrix kernels for shared memory and message passing systems, Tech. Rep. CMU-CS-97-178, Carnegie Mellon University, School of Computer Science (1997).

[9] L. Oliker, R. Biswas, P. Husbands, X. Li, Effects of ordering strategies and programming paradigms on sparse matrix computations, Siam Review 44 (3) (2002) 373–393.

[10] S. Romero, L. F. Romero, E. L. Zapata, Fast cloth simulation with parallel computers, in: Proc. 6th International Euro-Par Conference on Parallel Processing, Lecture Notes In Computer Science, 2000, pp. 491–499.

[11] R. Lario, C. Garcia, M. Prieto, F. Tirado, Rapid parallelization of a multilevel cloth simulator using OpenMP, in: Proc. Third European Workshop on OpenMP, 2001, pp. 21–29.

[12] 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.

[13] F. Zara, F. Faure, J.-M. Vincent, Physical cloth animation on a PC cluster, in: Proc. of Fourth Eurographics Workshop on Parallel Graphics and Visualization, 2002, pp. 105–112.

[14] 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) (2004) 173–180.

[15] 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, pp. 119–126.

[16] B. Thomaszewski, W. Blochinger, Parallel simulation of cloth on distributed memory architectures, in: Proc. of Eurographics Symposium on Parallel Graphics and Visualization, 2006, pp. 35–42.

[17] A. Mujahid, K. Kakusho, M. Minoh, Y. Nakashima, S. Mori, S. Tomita, Simulating realistic force and shape of virtual cloth with adaptive meshes and its parallel implementation in OpenMP, in: Proc. of Intl. Conf. on Parallel and Distributed Computing and Networks (PDCN2004), 2004, pp. 386–391.

[18] P. G. Ciarlet, Mathematical Elasticity. Vol. I, North-Holland Publishing Co., 1992.

[19] O. Etzmuß, M. Keckeisen, W. Straßer, A fast finite element solution for cloth modelling, in: Proc. 11th Pacific Conference on Computer Graphics and Applications, 2003, pp. 244–251.

[20] 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.

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

[22] J. Mezger, S. Kimmerle, O. Etzmuß, Hierarchical techniques in collision detection for cloth animation, Journal of WSCG 11 (2) (2003) 322–329.

[23] R. Bridson, R. P. Fedkiw, J. Anderson, Robust treatment of collisions, contact, and friction for cloth animation, in: Proc. of ACM SIGGRAPH, 2002, pp. 594–603.

[24] B. C. Lee, R. W. Vuduc, J. W. Demmel, K. A. Yelick, Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply, in: Proc. of International Conference on Parallel Processing (ICPP'04), 2004, pp. 169–176.

[25] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, Journal of Parallel and Distributed Computing 48 (1) (1998) 96–129.

[26] M. Hauth, O. Etzmuß, A high performance solver for the animation of deformable objects using advanced numerical methods, Computer Graphics Forum 20 (3) (2001) 319–328.

[27] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, B. Lee, Performance optimizations and bounds for sparse matrix-vector multiply, in: Proc. of ACM/IEEE Supercomputing, 2002, p. 26.

[28] R. Bridson, S. Marino, R. Fedkiw, Simulation of clothing with folds and wrinkles, in: Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA 2003), 2003, pp. 28–36.

[29] W. Blochinger, W. Küchlin, C. Ludwig, A. Weber, An object-oriented platform for distributed high-performance Symbolic Computation, Mathematics and Computers in Simulation 49 (3) (1999) 161–178.

[30] K. H. Randall, Cilk: Efficient multithreaded computing, Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science (Jun. 1998).

[31] J. M. Mellor-Crummey, M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Transactions on Computer Systems 9 (1) (1991) 21–65.

[32] M. M. Michael, M. L. Scott, Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, in: Proc. of the 15th ACM Symposium on Principles of Distributed Computing, 1996, pp. 267–275.

[33] Advanced Micro Devices, AMD64 Architecture Programmers Manual Volume 3: General-Purpose and System Instructions (2007).

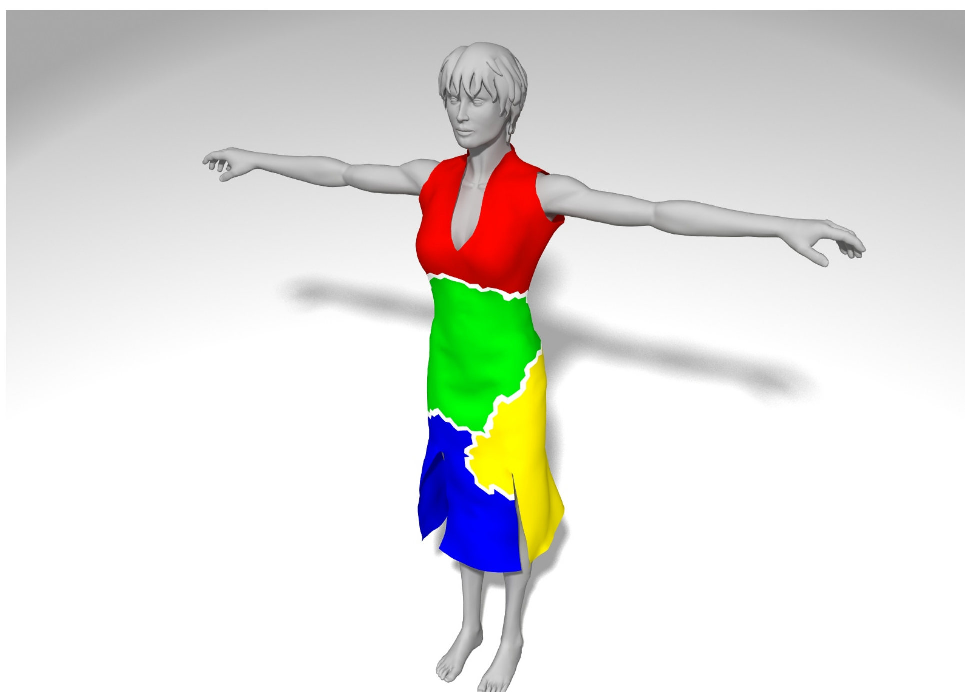

Fig. B.1. A snapshot from test scene 1a. A woman wearing a dress is simulated under the influence of gravity. The dress is comprised of slightly more than 4,500 vertices while the avatar consists of 27,000 vertices. Different colours have been used to indicate problem decomposition of the time integration stage.

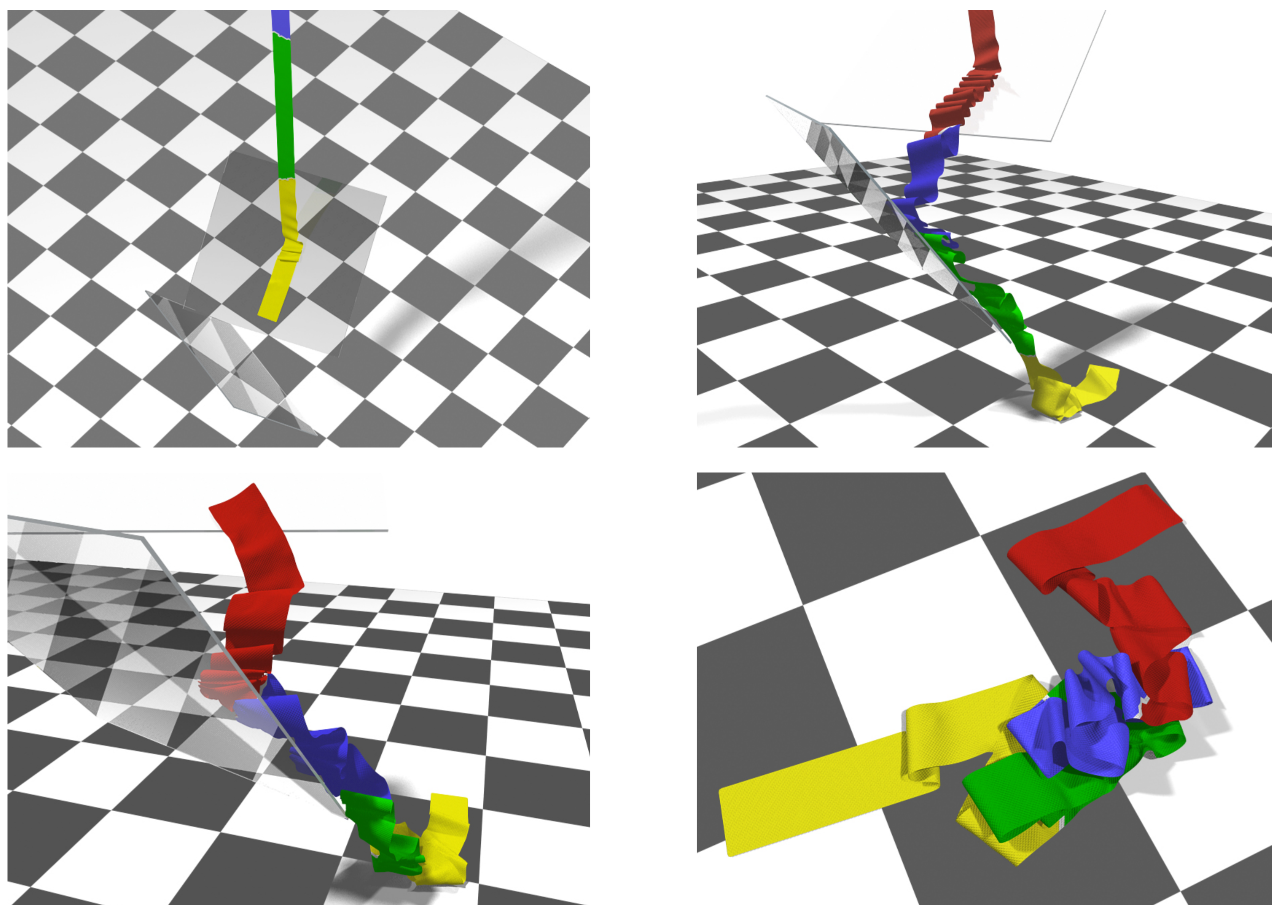

Fig. B.2. Four shots from the second test scene. A long (0.5m $\times$ 20.0m) ribbon consisting of 4,141 vertices falls on two slightly inclined planes and slides onto the floor. Due to surface friction complex folds are formed as it slides over the planes. This again leads to complicated self-collisions which are handled flawlessly by our parallel collision handling algorithm. Different colours have been used to indicate problem decomposition of the time integration stage.